Contents

- Preface
  - Who this book is for
  - What this book covers
  - To get the most out of this book
  - Get in touch
1. Welcome to Beginning C++ Game Programming Third Edition!
   - The games we will build
     - Timber!!!
     - Pong
     - Zombie Arena
     - Platform game
   - Why you should learn game programming using C++ in 2024
     - SFML
     - Microsoft Visual Studio
     - What about Mac and Linux?
     - Installing Visual Studio 2022
   - Setting up SFML
   - Creating a new project in Visual Studio 2022
     - Configuring the project properties
   - Planning Timber!!!
   - The project assets
     - Making your own sound FX
     - Adding the assets to the project
     - Exploring the assets
   - Understanding screen and internal coordinates
   - Getting started with coding the game
     - Making code clearer with comments
     - The main function
     - Presentation and syntax
     - Returning values from a function
     - Running the game
   - Opening a window using SFML
     - Including SFML features
     - OOP, classes, and objects

        - Using namespace sf
        - SFML VideoMode and RenderWindow
        - Running the game
    - The game loop
        - while loops
        - C-style code comments
        - Input, update, draw, repeat
        - Detecting a key press
        - Clearing and drawing the scene
        - Running the game
    - Drawing the game's background
        - Preparing the sprite using a texture
        - Double buffering the background sprite
        - Running the game
    - Handling errors
        - Configuration errors
        - Compile errors
        - Link errors
        - Bugs
    - Summary
    - Frequently asked questions
2. Variables, Operators, and Decisions: Animating Sprites
    - Learning all about C++ variables
        - Types of variables
            - User-defined types
        - Declaring and initializing variables
            - Declaring variables
            - Initializing variables
            - Declaring and initializing in one step
            - Constants
            - Uniform initialization
            - Declaring and initializing user-defined types
    - Seeing how to manipulate the variables
        - C++ arithmetic and assignment operators
        - Getting things done with expressions
            - Assignment
            - Increment and decrement

- Adding clouds, a buzzing bee, and a tree
  - Preparing the tree
  - Preparing the bee
  - Preparing the clouds
  - Drawing the tree, the bee, and the clouds
- Random numbers
  - Generating random numbers in C++
- Making decisions with if and else
  - Logical operators
  - C++ if and else
  - If they come over the bridge, shoot them!
  - Else do this instead
  - Reader challenge
- Timing
  - The frame rate problem
  - The SFML frame rate solution
- Moving the clouds and the bee
  - Giving life to the bee
  - Blowing the clouds
- Summary
- Frequently Asked Questions

3. C++ Strings, SFML Time: Player Input and HUD
   - Pausing and restarting the game
   - C++ strings
     - Declaring strings
     - Assigning a value to strings
     - String Concatenation
     - Getting the string length
     - Manipulating strings another way with StringStream
     - SFML Text and SFML Font
   - Adding a score and a message
   - Adding a time-bar
   - Summary
   - Frequently asked questions
4. Loops, Arrays, Switch, Enumerations, and Functions: Implementing Game Mechanics
   - Loops

- while loops
- Breaking out of a loop
- for loops
- Arrays
  - Declaring an array
  - Initializing the elements of an array
    - Quickly initializing the elements of an array
  - What do these arrays really do for our games?
- Making decisions with switch
- Class enumerations
- Getting started with functions
  - Who designed all this weird and frustrating syntax and why is it the way it is?
  - Function return types
  - Function names
  - Function parameters
  - The function body
  - Function prototypes
  - Organizing functions
  - Function scope
  - A final word on functions – for now
- Growing the branches
  - Preparing the branches
  - Updating the branch sprites in each frame
  - Drawing the branches
  - Moving the branches
- Summary
- Frequently asked questions

5. Collisions, Sound, and End Conditions: Making the Game Playable
   - Preparing the player (and other sprites)
   - Drawing the player and other sprites
   - Handling the player's input
     - Handling setting up a new game
     - Detecting the player chopping
     - Detecting a key being released
     - Animating the chopped logs and the axe
   - Handling death

- Simple sound effects
  - How SFML sound works
  - When to play the sounds
  - Adding the sound code
- Improving the game and code
- Summary
- Frequently asked questions

6. Object-Oriented Programming – Starting the Pong Game
   - Object-oriented programming
     - Encapsulation
     - Polymorphism
     - Inheritance
     - Why use OOP?
     - What exactly is a class?
   - The theory of a Pong bat
     - Declaring the class, variables, and functions
     - The class function definitions
     - Using an instance of a class
   - Creating the Pong project
   - Coding the Bat class
     - Coding Bat.h
     - Constructor functions
     - Continuing with the Bat.h explanation
     - Coding Bat.cpp
   - Using the Bat class and coding the main function
   - Summary
   - Frequently asked questions
7. AABB Collision Detection and Physics – Finishing the Pong Game
   - Coding the Ball class
   - Using the Ball class
   - Collision detection and scoring
   - Running the game
   - Learning about the C++ spaceship operator
   - Summary
   - Frequently asked questions
8. SFML Views – Starting the Zombie Shooter Game
   - Planning and starting the Zombie Arena game

        - Creating a new project
        - The project assets
        - Exploring the assets
        - Adding the assets to the project
    - OOP and the Zombie Arena project
    - Building the player – the first class
        - Coding the Player class header file
        - Coding the Player class function definitions
    - Controlling the game camera with SFML View
    - Starting the Zombie Arena game engine
    - Managing the code files
    - Starting to code the main game loop
    - Summary
    - Frequently asked questions
9. C++ References, Sprite Sheets, and Vertex Arrays
    - Understanding C++ references
        - Summarizing references
    - SFML vertex arrays and sprite sheets
        - What is a sprite sheet?
    - What is a vertex array?
        - Building a background from tiles
        - Building a vertex array
        - Using the vertex array to draw
    - Creating a randomly generated scrolling background
    - Using the background
    - Summary
    - Frequently asked questions
10. Pointers, the Standard Template Library, and Texture Management
    - Learning about pointers
        - Pointer syntax
        - Declaring a pointer
        - Initializing a pointer
        - Reinitializing pointers
        - Dereferencing a pointer
        - Pointers are versatile and powerful
            - Dynamically allocated memory
            - Passing a pointer to a function

        - Declaring and using a pointer to an object
    - Pointers and arrays
    - Summary of pointers
- Learning about the Standard Template Library
    - What is a vector?
        - Declaring a vector
        - Adding data to a vector
        - Accessing data in a vector
        - Removing data from a vector
        - Checking the size of a vector
        - Looping/iterating through the elements of a vector
    - What is a map?
        - Declaring a map
        - Adding data to a map
        - Finding data in a map
        - Removing data from a map
        - Checking the size of a map
        - Checking for keys in a map
        - Looping/iterating through the key-value pairs of a map
    - The auto keyword
    - STL summary
- Summary
- Frequently asked questions

11. Coding the TextureHolder Class and Building a Horde of Zombies
- Implementing the TextureHolder class
    - Coding the TextureHolder header file
    - Coding the TextureHolder function definitions
    - What have we achieved with TextureHolder?
- Building a horde of zombies
    - Coding the Zombie.h file
    - Coding the Zombie.cpp file
    - Using the Zombie class to create a horde
    - Bringing the horde to life (or back to life)
- Using the TextureHolder class for all textures
    - Changing the way the background gets its textures
    - Changing the way the Player class gets its texture
- Summary

- Frequently asked questions

12. Collision Detection, Pickups, and Bullets
    - Coding the Bullet class
        - Coding the Bullet header file
        - Coding the Bullet source file
        - Coding the shoot function
            - Calculating the gradient in the shoot function
            - Making the gradient positive in the shoot function
            - Calculating the ratio between X and Y in the shoot function
            - Finishing the shoot function explanation
        - More bullet functions
        - The Bullet class's update function
    - Making the bullets fly
        - Including the Bullet class
        - Control variables and the bullet array
        - Reloading the gun
        - Shooting a bullet
        - Updating the bullets in each frame
        - Drawing the bullets in each frame
        - Giving the player a crosshair
    - Coding a class for pickups
        - Coding the Pickup header file
        - Coding the Pickup class function definitions
    - Using the Pickup class
    - Detecting collisions
        - Has a zombie been shot?
        - Has the player been touched by a zombie?
        - Has the player touched a pickup?
    - Summary
    - Frequently asked questions
13. Layering Views and Implementing the HUD
    - Adding all the Text and HUD objects
    - Updating the HUD
    - Drawing the HUD, home, and level-up screens
    - Summary
14. Sound Effects, File I/O, and Finishing the Game

- Saving and loading the high score
- Preparing sound effects
- Allowing the player to level up and spawning a new wave
- Restarting the game
- Playing the rest of the sounds
    - Adding sound effects while the player is reloading
    - Making a shooting sound
    - Playing a sound when the player is hit
    - Playing a sound when getting a pickup
    - Making a splat sound when a zombie is shot
- Summary
- Frequently asked questions

15. Run!
    - About the game
    - Creating the project
    - Coding the main function
    - Handling input
    - Coding the Factory class
    - Advanced OOP: inheritance and polymorphism
        - Inheritance
        - Extending a class
        - Polymorphism
        - Abstract classes: virtual and pure virtual functions
    - Design patterns
    - Entity Component System pattern
        - Why lots of diverse object types are hard to manage
        - Using a generic GameObject for better code structure
        - Prefer composition over inheritance
        - Factory pattern
        - C++ smart pointers
            - Shared pointers
            - Unique pointers
        - Casting smart pointers
        - Coding the GameObject class
        - Coding the Component class
        - Coding the Graphics class
        - Coding the Update class

        - Running the code
        - What next?
    - Summary
16. Sound, Game Logic, Inter-Object Communication, and the Player
    - Coding the SoundEngine class
    - Code the Game logic
        - Coding the LevelUpdate class
    - Coding the player: Part 1
        - Coding the PlayerUpdate class
        - Coding the PlayerGraphics class
    - Coding the factory to use all our new classes
        - Remembering the texture coordinates
    - Running the game
    - Summary
17. Graphics, Cameras, Action
    - Cameras, draw calls, and SFML View
    - Coding the camera classes
        - Coding the CameraUpdate class
        - Coding the CameraGraphics class part 1
        - The SFML View class
        - Coding the CameraGraphics class part 2
    - Adding camera instances to the game
    - Running the game
    - Summary
18. Coding the Platforms, Player Animations, and Controls
    - Coding the platforms
        - Coding the PlatformUpdate class
            - Coding the update function for the PlatformUpdate class
        - Coding the PlatformGraphics class
        - Building some platforms in the factory
    - Running the game
    - Adding functionality to the player
        - Coding the player controls
    - Running the game
    - Coding the Animator class
    - Coding the player animations

- Running the game
- Summary

19. Building the Menu and Making It Rain
    - Building an interactive menu
        - Coding the MenuUpdate class
        - Coding the MenuGraphics class
        - Building a menu in the factory
    - Running the game
    - Making it rain
        - Coding the RainGraphics class
        - Making it rain in the factory
    - Running the game
    - Summary
20. Fireballs and Spatialization
    - What is spatialization?
        - Emitters, attenuation, and listeners
    - Handling spatialization using SFML
    - Upgrading the SoundEngine class
    - Fireballs
        - Coding the FireballUpdate class
        - Coding the FireballGraphics class
            - Coding FireballGraphics.h
            - Coding FireballGraphics.cpp
        - Building some fireballs in the factory
    - Running the code
    - Summary
21. Parallax Backgrounds and Shaders
    - Learning about OpenGL, shaders, and GLSL
        - The programmable pipeline and shaders
        - Coding a hypothetical fragment shader
        - Coding a hypothetical vertex shader
    - Finishing the CameraGraphics class
        - Breaking up the new draw code
    - Coding a shader for the game
    - Running the completed game
    - Summary
    - Further reading

Other Books You May Enjoy
Index

Beginning C++ Game Programming

Third Edition

Learn C++ from scratch by building fun games

John Horton

Beginning C++ Game Programming

Third Edition

Copyright © 2024 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Senior Publishing Product Manager: Larissa Pinto

Acquisition Editor – Peer Reviews: Jane Dsouza

Project Editor: Meenakshi Vijay

Content Development Editor: Shikha Parashar

Copy Editor: Safis Editing

Technical Editor: Simanta Rajbangshi

Proofreader: Safis Editing

Indexer: Hemangini Bari

Presentation Designer: Rajesh Shirsath

Developer Relations Marketing Executive: Sohini Ghosh

First published: October 2016

Second edition: October 2019

Third edition: May 2024

Production reference: 1240524

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul’s Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83508-174-7

www.packt.com

Contributors

About the author

John Horton is a programming and gaming enthusiast based in the UK.

Dedicated to two brothers, Ray and Barry, for your guidance, example, and support.

About the reviewer

Yoan Rock is a 26-year-old developer with over 4 years of experience in the gaming industry. With a background in C++ software engineering, Yoan's expertise lies in C++ programming within the gaming industry, particularly in utilizing Unreal Engine and sometimes Blueprints to create immersive experiences.

During his tenure at Limbic Studio, Yoan contributed significantly to the development of *Park Beyond*, an AAA-released game where players create and manage their own theme park. He excelled in gameplay development, bug fixing, and fostering effective communication among team members.

Yoan later collaborated with Chillchat on *Primorden*, a multiplayer project using Unreal Engine 5 and the Game play Ability system, where he played a key role in implementing game mechanics, monster abilities, and AI behavior trees.

At Game Atelier, Yoan led UI development for an unannounced project, showcasing his proficiency in crafting immersive user experiences using Unreal Engine 5.3, Common UI, and, of course, UMG.

Currently, Yoan is part of an exciting project with Blacksheep, contributing to an ambitious, unannounced venture. Always eager to innovate, Yoan stays updated with industry trends and is exploring Unreal Engine 5.3 for personal projects.

Preface

Always dreamed of creating your own games? With the third edition of *Beginning C++ Game Programming*, you can turn that dream into reality! This beginner-friendly guide is updated and improved to include the latest features of **VS 2022**, **SFML**, and modern **C++20** programming techniques. You will get a fun introduction to game programming by building four fully playable games of increasing complexity. You'll build clones of popular games such as Timberman, Pong, a Zombie survival shooter, and an endless runner.

The book starts by covering the basics of programming. You'll study key C++ topics, such as **object-oriented programming** (**OOP**) and C++ pointers, and get acquainted with the **Standard Template Library** (**STL**). The book helps you learn about collision detection techniques and game physics by building a Pong game. As you build games, you'll also learn exciting game programming concepts such as vertex arrays, directional sound (spatialization), OpenGL programmable shaders, spawning objects, and much more. You'll dive deep into game mechanics and implement input handling, levelling up a character, and simple enemy AI. Finally, you'll explore game design patterns to enhance your C++ game programming skills.

By the end of the book, you'll have gained the knowledge you need to build your own games with exciting features from scratch.

Who this book is for

This book is perfect for you if you have no C++ programming knowledge, you need a beginner-level refresher course, or you want to learn how to build games or just use games as an engaging way to learn C++.

Whether you aspire to publish a game (perhaps on Steam) or just want to impress friends with your creations, you'll find this book useful.

What this book covers

Chapter 1, *Welcome to Beginning C++ Game Programming, Third Edition*: This chapter outlines the journey to writing exciting games for PC using C++ and the OpenGL powered **SFML**. This third edition has an overwhelming focus on improving and expanding upon what you will learn in game programming. All the C++ basics from variables in the beginning, through loops, object-oriented programming, the **Standard Template Library**, SFML features, and newer C++ possibilities, have been added to and expanded upon. By the end of this book, you will not only have four playable games but also have a deep and solid grounding in C++.

Chapter 2, *Variables, Operators, and Decisions: Animating Sprites*: In this chapter, we will do quite a bit more drawing on the screen. We will animate some clouds that travel at a random height and a random speed across the background and a bee that does the same in the foreground. To achieve this, we will need to learn some more of the basics of C++. We will be learning how C++ stores data with variables as well as how to manipulate those variables with the C++ operators and how to make decisions that branch our code on different paths based on the value of variables. Once we have learned all this, we will be able to reuse our knowledge about the SFML `Sprite` and `Texture` classes to implement our cloud and bee animations.

Chapter 3, *C++ Strings, SFML Time, Player Input, and HUD*: In this chapter, we will spend around half the time learning how to manipulate text and display it on the screen and the other half looking at timing and how a visual time bar can inform the player and create a sense of urgency in the game.

Chapter 4, *Loops, Arrays, Switch, Enumerations, and Functions – Implementing Game Mechanics*: This chapter probably has more C++ information than any other chapter in the book. It is packed with fundamental concepts that will move our understanding on enormously. It will also begin to shed light on some of the murky areas we have been skipping over a little bit, like functions, the game loop, and loops in general.

Chapter 5, *Collisions, Sound, and End Conditions: Making the Game Playable*: This is the final phase of the first project. By the end of this chapter, you will have your first completed game. Once you have Timber!!! up and running, be sure to read the last section of this chapter as it will suggest ways to make the game better. Here is what we will cover in this chapter: adding the rest of the sprites, handling the player input, animating the flying log, handling death, adding sound effects, adding features, and improving Timber!!!.

Chapter 6, *Object-Oriented Programming – Starting the Pong Game*: In this chapter, there's a little bit of theory, but the theory will give us the knowledge that we need to start using **object-oriented programming** (**OOP**). OOP helps us organize our code into human-recognizable structures and handle complexity. We will not waste any time in putting that theory to good use as we will use it to code the next project, a Pong game. We will get to look behind the scenes at how we can create new C++ types that we can use as objects. We will achieve this by coding our first class. To get started, we will look at a simplified Pong game scenario so that we can learn about some class basics, and then we will start again and code a Pong game for real using the principles we have learned.

Chapter 7, *AABB Collision Detection and Physics – Finishing the Pong Game*: In this chapter, we will code our second class. We will see that

although the ball is obviously quite different from the bat, we will use the exact same techniques to encapsulate the appearance and functionality of a ball inside a `Ball` class, just like we did with the bat and the `Bat` class. We will then add the finishing touches to the Pong game by coding some collision detection and scorekeeping. This might sound complicated but as we are coming to expect, SFML will make things much easier than they otherwise would be.

Chapter 8, *SFML Views – Starting the Zombie Shooter Game*: In this project, we will be making even more use of **OOP**, and to a powerful effect. We will also be exploring the SFML `View` class. This versatile class will allow us to easily divide our game up into layers for different aspects of the game. In the Zombie Shooter project, we will have a layer for the HUD and a layer for the main game. This is necessary because the game world expands each time the player clears a wave of zombies. Eventually, the game world will be bigger than the screen and the player will need to scroll. The use of the `View` class will prevent the text of the HUD from scrolling with the background.

Chapter 9, *C++ References, Sprite Sheets, and Vertex Arrays*: In *Chapter 4*, *Loops, Arrays, Switch, Enumerations, and Functions – Implementing Game Mechanics*, we talked about scope. This is the concept that variables declared in a function or inner block of code only have scope (that is, can be *seen* or used) in that function or block. Using only the C++ knowledge we have currently, this can cause a problem. What do we do if we need to work on a few complex objects that are needed in the `main` function? This could imply all the code must be in the `main` function.

In this chapter, we will explore C++ **references**, which allow us to work on variables and objects that are otherwise out of scope. In addition to this,

these references will help us avoid having to pass large objects between functions, which is a slow process. It is slow because each time we do this, a copy of the variable or object must be made.

Armed with this new knowledge of references, we will look at the SFML `VertexArray` class, which allows us to build up a large image that can be quickly and efficiently drawn to the screen using multiple parts in a single image file. By the end of this chapter, we will have a scalable, random, scrolling background that's been made using references and a `VertexArray` object.

Chapter 10, *Pointers, the Standard Template Library, and Texture Management*: In this chapter, we will learn a lot as well as get plenty done in terms of the game in this chapter. We will first learn about the fundamental C++ topic of **pointers**. Pointers are variables that hold a memory address. Typically, a pointer will hold the memory address of another variable. This sounds a bit like a reference, but we will see how they are much more powerful and use a pointer to handle an ever-expanding horde of zombies.

We will also learn about the **Standard Template Library** (**STL**), which is a collection of classes that allow us to quickly and easily implement common data management techniques.

Chapter 11, *Coding the TextureHolder Class and Building a Horde of Zombies*: Now that we have understood the basics of the STL, we will be able to use that new knowledge to manage all the textures from the game because if we have 1,000 zombies, we don't really want to load a copy of a zombie graphic into the GPU for each and every one.

We will also dig a little deeper into OOP and use a **static** function, which is a function of a class that can be called without an instance of the class. At

the same time, we will see how we can design a class to ensure that only one instance can ever exist. This is ideal when we need to guarantee that different parts of our code will use the same data.

Chapter 12, *Collision Detection, Pickups, and Bullets*: So far, we have implemented the main visual aspects of our game. We have a controllable character running around in an arena full of zombies that chase them. The problem is that they don't interact with each other. A zombie can wander right through the player without leaving a scratch. We need to detect collisions between the zombies and the player.

If the zombies are going to be able to injure and eventually kill the player, it is only fair that we give the player some bullets for their gun. We will then need to make sure that the bullets can hit and kill the zombies.

At the same time, if we are writing collision detection code for bullets, zombies, and the player, it would be a good time to add a class for health and ammo pickups as well.

Here is what we will do and the order in which we will cover things in this chapter: shooting bullets, adding a crosshair and hiding the mouse pointer, spawning pickups, and detecting collisions

Chapter 13, *Layering Views and Implementing the HUD*: In this chapter, we will get to see the real value of SFML `Views`. We will add a selection of SFML `Text` objects and manipulate them as we did before in the Timber!!! project and the Pong project. What's new is that we will draw the HUD using a second View instance. This way, the HUD will stay neatly positioned over the top of the main game action, regardless of what the background, player, zombies, and other game objects are doing.

Chapter 14, *Sound Effects, File I/O, and Finishing the Game*: We are nearly done with this project. This short chapter will demonstrate how we can easily manipulate files stored on the hard drive using the C++ standard library, and we will also add sound effects. Of course, we know how to add sound effects, but we will discuss exactly where the calls to the `play` function will go in the code. We will also tie up a few loose ends to make the game complete. In this chapter, we will cover the following topics: saving and loading the hi-score using file input and file output, adding sound effects, allowing the player to level up, and spawning a new wave.

Chapter 15, *Run!*: Welcome to the final project. Run, Run is an endless runner where the objective of the player is to stay ahead of the disappearing platforms that are catching them up from behind. In this project, we will learn loads of new game programming techniques and even more C++ topics to implement those techniques. Perhaps the best improvement this game will have over the previous games is that it will be way more object oriented than any of the others. There will be many, many more classes than any of the preceding projects but most of the code files for these classes will be short and uncomplicated. Furthermore, we will build a game where the functionality and appearance of all the in-game objects is pushed out to classes, leaving the main game loop unchanged regardless of what the `GameObjects` do. This is powerful because it means you can make a hugely varied game just by designing new standalone components (classes) that describe the behavior and appearance of the required game entity. This means you can use the same code structure for a completely different game of your own design. But there is way more to come than just this. Read on for details.

Chapter 16, *Sound, Game Logic, Inter-Object Communication, and the Player*: In this chapter, we will quickly implement our game's sound. We

have done this before, so it won't be hard. In fact, in just half a dozen lines of code, we will also add music to our sound features. Later in the project, but not in this chapter, we will add directional (spatialized) sound.

In this chapter, we will wrap all our sound-related code into a single class called `SoundEngine`. Once we have some noise, we will then move on to get started on the player. We will achieve the entire player character functionality just by adding two classes: one that extends `Update` and one that extends `Graphics`. This creation of new game objects by extending these two classes will be how we do almost everything else for the entire game. We will also see the simple way that objects communicate with each other using pointers.

Chapter 17, *Graphics, Camera, Action*: In this chapter, we will talk in depth about the way the graphics will work in this project. As we will be coding the cameras that do the drawing in this chapter, now seems like a good time to talk about the graphics too. If you looked in the `graphics` folder, there is just one graphic. Furthermore we are not calling `window.draw` at any point in our code so far. We will discuss why draw calls should be kept to a minimum as well as implement our `Camera` classes that will handle this for us. Finally, in this chapter, we will be able to run the game and see the cameras in action, including the main view, the radar view, and the timer text.

Chapter 18, *Coding the Platforms, Player Animations, and Controls*: In this chapter, we will code the platforms and the player animation and controls. In my opinion, we have done the hard work already and most of what follows has a much higher reward-to-effort ratio. Hopefully this chapter will be interesting as we will see how the platforms will ground the player and enable them to run, as well as seeing how we loop through the frames

of animation to create a smooth running effect for the player. We will do the following: coding the platforms, adding functionality to the player, coding the `Animator` class, coding the animations, and adding a smooth running animation to the player.

Chapter 19, *Building the Menu and Making It Rain*: In this chapter, we will implement two significant features. One is a menu screen to keep the player informed of their options for starting, pausing, restarting, and quitting the game. The other job will be to create a simple raining effect. You could argue the raining effect isn't necessary, even that it doesn't fit the game, but it is easy, fun, and a good trick to learn. What you should expect by now, and yet is still perhaps the most interesting aspect of this chapter, is how we will achieve both these objectives by coding classes derived from `Graphics` and `Update`, composing them in `GameObject` instances, and they will just work alongside all our other game entities.

Chapter 20, *Fireballs and Spatialization*: In this chapter, we will be adding all the sound effects and the HUD. We have done this in two of the previous projects, but we will do things a bit differently this time. We will explore the concept of sound **spatialization** and how SFML makes this complicated concept nice and easy. In addition, we will build a HUD class to encapsulate our code that draws information to the screen.

Chapter 21, *Parallax Backgrounds and Shaders*: This is the last chapter and our last opportunity to work on our game. It will be fully playable with all the features by the end. Here is what we will do to wrap up the Run game. We will learn a bit more about OpenGL, shaders, and the **Graphics Library Shading Language** (**GLSL**), finish the `CameraGraphics` class by implementing a scrolling background and shader, a code a shader by using someone else's code, and finally run the completed game

To get the most out of this book

There are no knowledge prerequisites for this book. You do not need to know how to program as the book takes you from zero knowledge to four playable games. It will help a little if you have played a few video games and you are determined to learn.

Download the example code files

The code bundle for the book is hosted on GitHub at https://github.com/PacktPublishing/Beginning-C-Game-Programming-Third-Edition. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://packt.link/gbp/9781835081747.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: “My main project directory is `D:\VS Projects\Timber`.”

A block of code is set as follows:

```
int playerScore = 0;
char playerInitial = 'J';
float valuePi = 3.141f;
bool isAlive = true;
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
// Make a tree sprite
Texture textureTree;
textureTree.loadFromFile("graphics/tree.png");
Sprite spriteTree;
spriteTree.setTexture(textureTree);
spriteTree.setPosition(810, 0);
while (window.isOpen())
{
```

Any command-line input or output is written as follows:

```
# cp /usr/src/asterisk-addons/configs/cdr_mysql.conf.sample
     /etc/asterisk/cdr_mysql.conf
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “Select **System info** from the **Administration** panel.”

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit http://www.packtpub.com/submit-errata, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit http://authors.packtpub.com.

Share your thoughts

Once you've read *Beginning C++ Game Programming, Third Edition*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don’t worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don’t stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:

https://packt.link/free-ebook/9781835081747

2. Submit your proof of purchase.
3. That’s it! We’ll send your free PDF and other benefits to your email directly.

1

Welcome to Beginning C++ Game Programming Third Edition!

Let's get started on your journey to writing exciting games for the PC using C++ and the OpenGL-powered **SFML**. This third edition has an overwhelming focus on improving and expanding upon what you will learn. All the C++ basics from variables in the beginning, through loops, object-oriented programming, the **Standard Template Library**, **SFML** features, and newer C++ possibilities have all been added to and expanded upon. By the end of this book, not only will you have four playable games but you will also have a deep and solid grounding in C++.

Here is what is coming up in this chapter

- First, we will look at the four games we will build across this book. The first game is the exact same as the previous edition and will help us learn the C++ basics, like **variables**, **loops**, and decision-making. The second and third are enhanced, modified, and refined from the previous edition, and the fourth is all new and, in my view, way better for playing and learning than the final two games of the previous edition put together.
- This next bit is mportant in which you will discover why you should learn game programming and perhaps any other programming genre

using C++. Using C++ to learn game development can be the best choice for so many reasons.

- Then, we can explore **SFML** and its relationship with C++.
- Nobody likes corporate evangelism, and you won't get any here, but there are good reasons to find out about **Microsoft Visual Studio** and why we will use it in this book.
- Next, it's time to set up the development environment. This is admittedly a slightly dull affair, but we will get through it in short order, step by step, and when you have done it once, you will never need to learn it again.
- We will then plan and prepare for the first game project, Timber!!!
- Moving on, we will write the first C++ code of this book and make a runnable first stage of the game that draws a pretty background – ooh! In the next chapter, things will advance and begin to move graphics around and what we learned in this chapter will stand us in good stead to make faster progress with our first game.
- Finally, we will cover how to handle any problems you might get as you learn C++ and game programming, such as configuration errors, compile errors, link errors, and bugs.

Of course, what you want to know first is what you are going to have to show for yourself by the end of this weighty tome. So, let's find out more about the games we will build.

You will find this chapter's source code in the GitHub repository: https://github.com/PacktPublishing/Beginning-C-Game-Programming-Third-Edition/tree/main/Timber

The games we will build

This journey will be smooth as we will learn about the fundamentals of the super-fast C++ language one step at a time, and then put this new knowledge to use by adding cool features to the four games we are going to build.

The following are our four projects for this book.

Timber!!!

The first game is an addictive, fast-paced clone of the hugely successful **Timberman**. Our game, **Timber**!!!, will introduce us to all the beginner basics of C++ while we build a genuinely playable game. Here is what our version of the game will look like when we are done and we have added a few last-minute enhancements:

Figure 1.1: Timber game

Timberman can be found at http://store.steampowered.com/app/398710/.

Pong

Pong was one of the first video games ever made. It is an excellent example of how the basics of game object animation, player input, and collision detection work. We will build a version of this simple retro game to explore the concept of classes and object-oriented programming. Here is what it will look like by the end of *Chapter 7*:

Figure 1.2: Pong game

The player will use the bat at the bottom of the screen and hit the ball back to the top of the screen. If you are interested, find out about Pong's history here: https://en.wikipedia.org/wiki/Pong.

Zombie Arena

Next, we will build a frantic, zombie survival shooter, not unlike the Steam hit *Over 9,000 Zombies!*, which you can find out more about at

http://store.steampowered.com/app/273500/. The player will have a machine gun and must fight off ever-growing waves of zombies. All this will take place in a randomly generated, scrolling world:

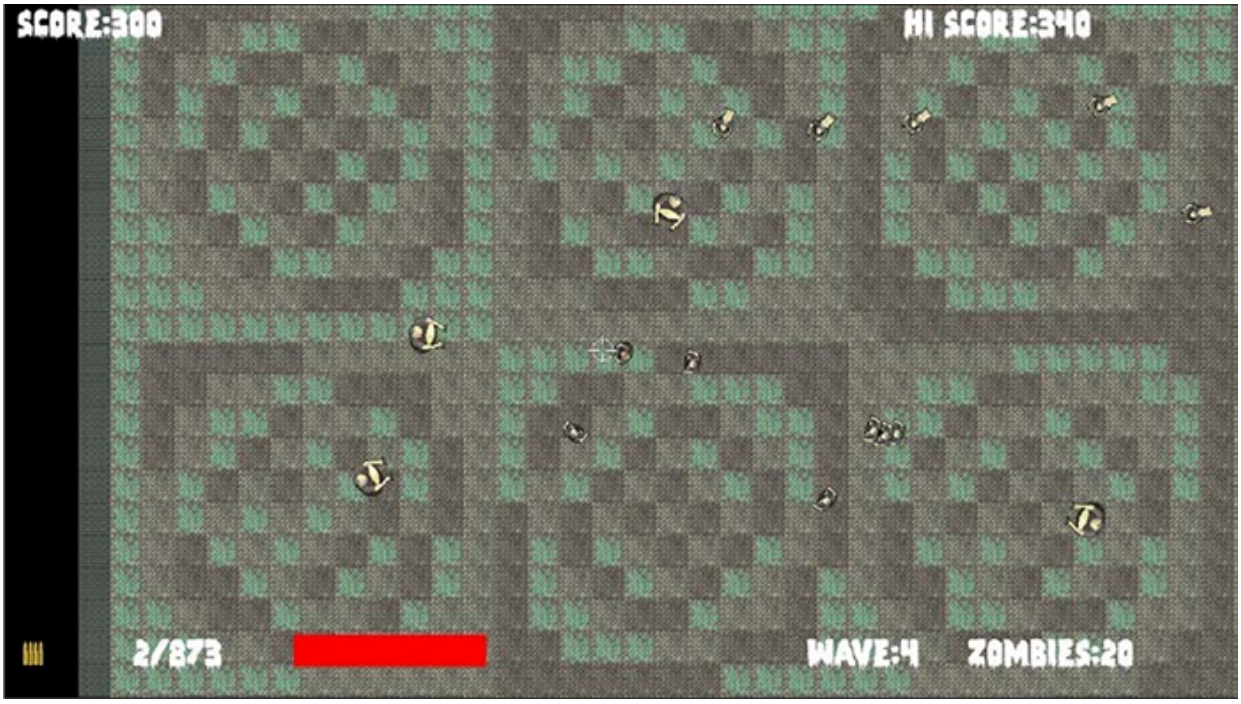


Figure 1.3: Zombie Arena game

To achieve this, we will learn about how object-oriented programming allows us to have a large **code base** (lots of code) that is easy to write and maintain. Expect exciting features such as hundreds of enemies, rapid-fire weaponry, pickups, and a character that can be leveled up after each wave.

Platform game

The final game is a platform game called **Run**. Run will be packed with more features enabled by the C++ skills we will have acquired and made easier by the great features of SFML. Take a look at the finished game below:

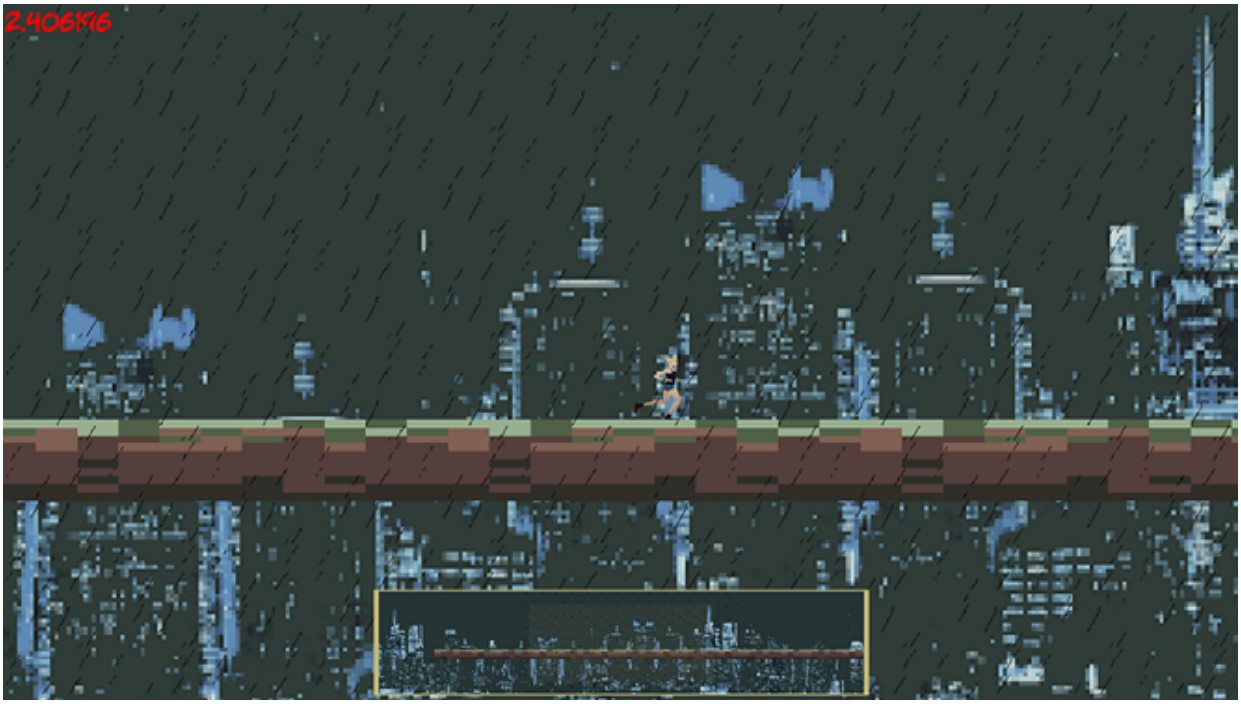


Figure 1.4: Platform game

Features include a photo-realistic shader background, parallax scrolling cityscape, spatialized (directional) sound, mini-map, animated player character, rain weather effect, music, pop-up menu, and more. Best of all, the final game will have a reusable code structure that you can use to invent and add your own features to.

Why you should learn game programming using C++ in 2024

The title above could also have read, “Why use game programming to learn C++…”, because C++, game programming, and beginners (in my view) are a perfect match. Let’s look at C++ in more detail while also staying focussed on games and beginners:

- **Speed**: C++ is known for its high performance and efficiency. In game development, performance is important. C++ allows you to write code that can run close to the native languages of both the CPU and the GPU, making it well suited for anything demanding, which includes games. This is achieved because C++ is turned into native executable instructions. This is just what we need when coding games with hundreds, thousands, or even hundreds of thousands of entities in it. In the final chapter, *Chapter 21*, we will see how C++ can interact directly with the GPU using shader programs.
- **Cross-platform development**: C++ works almost everywhere, meaning you can write code that can be compiled and run on various platforms without significant modifications. This book will focus on Windows but everything we learn and write in this book, with minor modifications, will work on macOS and Linux. C++ itself is also used extensively in next-gen console game development and can even be useful on mobile. Compiled means translating our C++ code into binary machine instructions for the CPU.
- **Lots of game engines and libraries**: Many game engines and libraries are written in C++ or provide C++ APIs. Learning C++ gives you access to the widest range of tools and resources for game development, such as **Unreal Engine**, as well as the fastest and best graphics libraries like **Vulcan**, **OpenGL**, **DirectX**, and **Metal**, as well as physics libraries like **Box2D**, UI tools like **IMGUI**, and networking libraries for co-op and multiplayer like **RakNet**, **Enet** and SFML's very own networking features.
- **Low-level control**: C++ provides low-level control over hardware resources, which is crucial for optimizing game performance. In game development, you may need to manage memory, optimize rendering

pipelines, and maintain control over the system your game is running on, and C++ offers the flexibility and power to do this. If managing memory and rendering pipelines sound ominous, then I can assure you that things will be fine. We introduce both these topics in a completely beginner-friendly manner in *Chapters 10* and *21*, respectively. Far from leaving you baffled, knowing how these powerful things can be controlled will leave you feeling powerful and in control of your programming destiny.

- **Documentation and support**: There is a thriving community around C++ game development, with numerous resources, tutorials, and forums available to help you learn and troubleshoot issues. If you have a C++ problem, I can guarantee you are not the first and a quick web search will almost always yield a solution. ChatGPT is an ace C++ problem solver, too.
- Learning C++ does have challenges but, taken a step at a time, is easily mastered. It is so rewarding to struggle over a problem and finally see it burst into an exciting gameplay feature when you get it right. Game development often involves seemingly difficult algorithms, data structures, and principles but C++ provides tools like the **Standard Template Library** (**STL**) and classes through **object-oriented programming** (**OOP**) to boil down complexity into manageable chunks. We will be covering OOP and STL in *Chapters 6* and *10*, respectively.
- **C++ is an industry standard**: It is because of everything we have just discussed that C++ is widely used in the game development industry. Familiarity with C++ can make it easier to collaborate with other developers, understand existing code bases, switch between game engines, and secure highly paid jobs in the industry.

Critics will say that C++ can have a steeper learning curve compared to some other programming languages and that if you're new to programming or game development, you might consider starting with a more beginner-friendly language like **C#** (for Unity development) or **Python** (for simple game projects) before diving into C++. There is some truth in this, but it is nowhere near as true as it used to be. C++ is constantly evolving, and numerous improvements to simplify learning and dramatically speed up development have been introduced in recent years. For example, new keywords like **auto**, intriguing-sounding logic operators like **spaceship**, as well as language constructs like lambdas, coroutines, and smart pointers, were introduced over the last 10 years, which dramatically simplify and speed up C++ development.

In summary, I would suggest that *not* learning C++ as a first language might be a mistake. And if you want to make learning as fun and rewarding as it possibly can get then learning with games is a no-brainer. Finally, if you want to be an indie game developer or work for a top game studio, unless you have some very specific other path in mind, C++ is the way to go.

But having just stated that C++ is so wonderful and has so many paths and libraries, why would we choose SFML?

SFML

SFML is the **Simple Fast Media Library**. It is not the only C++ library for games and multimedia. It is possible to make an argument to use other libraries, but SFML seems to come through for me every time. Firstly, it is written using object-oriented C++. The benefits of object-oriented C++ are numerous, and you will experience them as you progress through this book.

SFML is also easy to get started with and is therefore a good choice if you are a beginner, yet at the same time, it has the potential to build the highest quality 2D games if you are a professional. So, a beginner can get started using SFML and not worry about having to start again with a new language/library as their experience grows. And if you want to build 3D games, C++ and SFML is a great introduction before moving on to Unreal Engine. As an aside, you can build 3D games with SFML and OpenGL but most SFML libraries are focused on 2D, as is this book.

Perhaps the biggest benefit is that most modern C++ programming uses OOP. Every C++ beginner's guide I have ever read uses and teaches OOP. OOP is the future (and the now) of coding in almost all languages, in fact. So why, if you're learning C++ from the beginning, would you want to do it any other way?

SFML has a library for just about anything you would ever want to do in a 2D game. SFML works using OpenGL, which can also make 3D games. OpenGL is the de facto free-to-use graphics library for games when you want it to run on more than one platform. When you use SFML, you are automatically using OpenGL.

SFML allows you to create the following:

- 2D graphics and animations, including scrolling game worlds.
- Sound effects and music playback, including high-quality directional sound.
- Input handling with a keyboard, mouse, and gamepad.
- Online multiplayer features.
- The same code can be compiled and linked on all major desktop operating systems, and mobile as well!

Extensive research has not uncovered any more suitable ways to build 2D games for PC with C++, even for expert developers, especially if you are a beginner and want to learn C++ in a fun gaming environment. C++, check. SFML, check. Surely we want to steer clear of the big controlling corporations, though, right?

Microsoft Visual Studio

Visual Studio is an **Integrated Development Environment** (**IDE**). Visual Studio provides a neat and well-featured interface that simplifies the game development process while keeping advanced features to hand. Beginners can benefit from features like code completion and syntax highlighting, which help streamline the process of learning C++. Visual Studio is almost unarguably the most advanced free-to-use IDE for C++. Microsoft gives it away, not to seek forgiveness for past transgressions but to get you hooked for the future using a premium version. So, let's take advantage of the free stuff for now.

Visual Studio offers a powerful debugger with features like breakpoints and call stacks. You can run your game in Visual Studio and have it pause at a point of your choosing. You can then inspect the values held by your code and step through execution a line at a time. This makes it easier for beginners to understand how their code works and troubleshoot otherwise near-impossible issues.

IntelliSense is Visual Studio's code suggestions and real-time error-checking tool. It can help those new to C++ learn the language more quickly by instantly highlighting mistakes and auto-suggesting what you might be trying to think of. This is not just a great learning tool for beginners but it is also a huge speed boost for professionals.

Visual Studio has a large and active community, and there are many tutorials, forums, and resources available to help beginners with their C++ and SFML projects in Visual Studio.

Visual Studio has many advanced features. As you grow in knowledge and ambition, Visual Studio can grow with you. Visual Studio integrates with popular **version control systems** (**VCSs**) like **Git**, making it easy to get started managing larger projects with multiple programmers. Visual Studio has performance profiling features that allow you to monitor the memory and CPU usage of your game and, therefore, improve and optimize your game.

Visual Studio is almost an industry standard. Being one of the most widely used IDEs for C++, Visual Studio has an enormous number of users. This means that beginners can find plenty of online help and tutorials specific to Visual Studio. As an aside, usually, the last place you will look for Visual Studio support will be Microsoft. Being knowledgeable with Visual Studio could be valuable to a future employer.

Visual Studio hides away the complexity of preprocessing, compiling, and linking. It wraps it all up with the press of a button. In addition to this, it provides a slick user interface for us to type our code into and manage what will become a large selection of code files and other project assets as well.

Having extolled the virtues of Visual Studio, it is also true that any game you can create with Visual Studio, you can also create with open-source tools. Visual Studio will just make your time as a beginner simpler, and if you decide to switch to a more ethical toolset at some point in the future, the change will be smoother than if you had gone straight to these other tools.

While there are advanced versions of Visual Studio that cost hundreds of dollars, we will be able to build all our games in the free Visual Studio 2022

Community edition. This is the latest free version of Visual Studio at the time of writing. If, when you are reading this, there is a newer version, I suggest using the newer version as Visual Studio tends to be highly backward compatible as well as maintaining a reasonably consistent user interface over the years. This means you can probably benefit from the new features and ease of availability of the latest version and still follow along with this book.

In the sections that follow, we will set up the development environment, beginning with a discussion on what to do if you are using Mac or Linux operating systems.

What about Mac and Linux?

The games that we will be making can be built to run on Windows, Mac, and Linux! The code we use will be identical for each platform. However, each version does need to be compiled and linked on the platform for which it is intended, and the tutorials will not be able to help with Mac and Linux.

It would be unfair to say, especially for complete beginners, that this book is perfectly suited for Mac and Linux users. Although, I guess, if you are an enthusiastic Mac or Linux user and you are comfortable with your operating system, you will likely succeed. Most of the extra challenges you will encounter will be in the initial setup of the development environment, SFML, and the first project.

To this end, I can highly recommend the following tutorials, which will hopefully replace the next 10 pages (approximately), up to the *Planning Timber!!!* section, at which point, this book will become relevant to all operating systems.

For Linux, read this to replace the next few sections: https://www.sfml-dev.org/tutorials/2.5/start-linux.php.

On Mac, read this tutorial to get started: https://www.sfml-dev.org/tutorials/2.5/start-osx.php.

Installing Visual Studio 2022

To start creating a game, we need to install Visual Studio 2022. Installing Visual Studio can be almost as simple as downloading a file and clicking a few buttons. There is nothing challenging about installing Visual Studio provided you choose the correct edition. I will clearly point out the correct edition at the point of choosing.

Note that, over the years, Microsoft is likely to change the name, appearance, and download page that's used to obtain Visual Studio. They might change the layout of the user interface and make the instructions that follow out of date. My experience, however, is that they try hard to maintain consistency between editions. Furthermore, the settings that we configure for each project are fundamental to C++ and SFML, so careful interpretation of the instructions that follow in this chapter will likely be possible, even if Microsoft does something radical to Visual Studio.

Let's get started with installing Visual Studio:

1. The first thing you need is a Microsoft account and your login details. If you have a Hotmail, Windows, Xbox, or MSN account, then you already have one. If not, you can sign up for a free one here: https://login.live.com/.

2. At the time of writing (May 2024), Visual Studio 2022 is the latest version, so hopefully, this chapter will be up to date for a while. To get started, visit https://visualstudio.microsoft.com/ and find the Visual Studio download. This next image shows what the page looks like at the time I visited the previous link:

Meet the Visual Studio family
Visual Studio
The most comprehensive IDE for .NET and C++ developers on Windows. Fully packed with a sweet array of tools and features to elevate and enhance every stage of software development.
Learn more →
Download Visual Studio
Community 2022
Professional 2022
Enterprise 2022
Free for individual developers, academic uses, and open source
Visual Studio Code
A standalone source code editor that runs on Windows, macOS, and Linux. The top pick for JavaScript and web developers, with extensions to support just about any programming language.
Learn more →
By using Visual Studio Code you agree to its license & privacy statement
Download Visual Studio Code

Figure 1.5: Downloading Visual Studio

3. Find the download for Visual Studio and choose **Community 2022** from the drop-down options. Note that editions other than Community are premium products that are not free and the Visual Studio Code option, also shown in this image, is not what we want for this book. Click the **Save** button and your download will begin.
4. When the download completes, run the download by double-clicking on it. After giving permission for Visual Studio to make changes to your computer, wait for the installer program to download some files and set up the next stage of the installation.
5. Shortly, you will be asked where you want to install Visual Studio. Choose a hard drive with at least 50 GB of storage. Various sources on

the web suggest you will get away with much less than 50 GB, but by the time you have started creating projects, 50 GB will make sure you have plenty of room for future development. When you are ready, locate the **Desktop development with C++** option and select it. Next, click the **Install** button. This step might take a while to complete.

Now, we are ready to turn our attention to SFML and then our first project.

Setting up SFML

This short tutorial will guide you through downloading the SFML files that allow us to include the functionality contained in the SFML library in our projects. In addition, we will see how we can use the SFML DLL files that will enable our compiled object code to run alongside SFML. To set up SFML, follow these steps:

1. Visit this link on the SFML website: `http://www.sfml-dev.org/download.php`. Click on the button that says **Latest stable version**, as shown here:

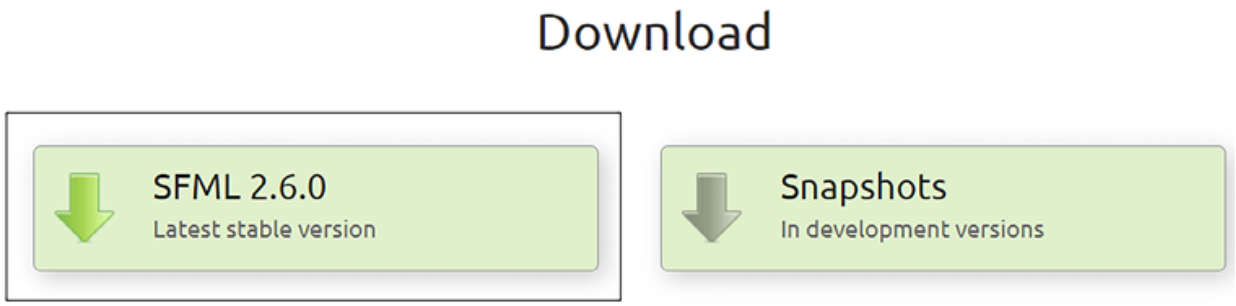


Figure 1.6: Downloading SFML 2.6

2. By the time you read this book, the latest version will almost certainly have changed. This won't matter if you do the next step just right. We want to download the 32-bit version. This might sound counter-intuitive

because you probably (most commonly) have a 64-bit PC. The reason we will download the 32-bit version is that 32-bit apps can run on both 32- and 64-bit machines. Furthermore, we need to get the Visual Studio 22 version. Click the **Download** button that's shown in the following screenshot:

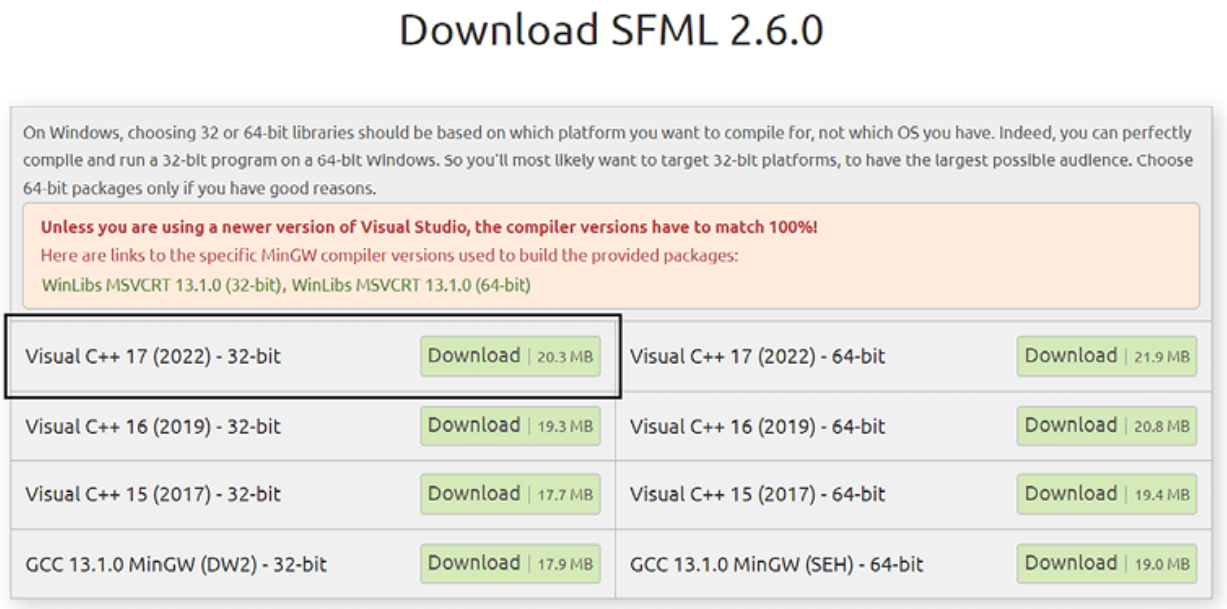

Download SFML 2.6.0
On Windows, choosing 32 or 64-bit libraries should be based on which platform you want to compile for, not which OS you have. Indeed, you can perfectly compile and run a 32-bit program on a 64-bit Windows. So you'll most likely want to target 32-bit platforms, to have the largest possible audience. Choose 64-bit packages only if you have good reasons.
Unless you are using a newer version of Visual Studio, the compiler versions have to match 100%!
Here are links to the specific MinGW compiler versions used to build the provided packages:
WinLibs MSVCRT 13.1.0 (32-bit), WinLibs MSVCRT 13.1.0 (64-bit)
Visual C++ 17 (2022) - 32-bit
Download | 20.3 MB
Visual C++ 17 (2022) - 64-bit
Download | 21.9 MB
Visual C++ 16 (2019) - 32-bit
Download | 19.3 MB
Visual C++ 16 (2019) - 64-bit
Download | 20.8 MB
Visual C++ 15 (2017) - 32-bit
Download | 17.7 MB
Visual C++ 15 (2017) - 64-bit
Download | 19.4 MB
GCC 13.1.0 MinGW (DW2) - 32-bit
Download | 17.9 MB
GCC 13.1.0 MinGW (SEH) - 64-bit
Download | 19.0 MB

Figure 1.7: Downloading SFML 17_22

3. When the download completes, create a folder at the root of the same drive where you installed Visual Studio and name it `SFML`. Also, create another folder at the root of the drive where you installed Visual Studio and call it `VS Projects`.
4. Finally, unzip the SFML download. Do this on your desktop. My file was called `SFML-2.6.0-windows-vc17-32-bit.zip` but yours may be different to reflect a newer version of SFML. When unzipping is complete, you can delete the `.zip` folder. You will be left with a single folder on your desktop. Its name will reflect the version of SFML that you downloaded. Double-click this folder to see its contents; I have a folder called `SFML-2.6.0`. Now double-click again into the folder.

The following screenshot shows what my SFML folder's content looks like. Yours should look the same.

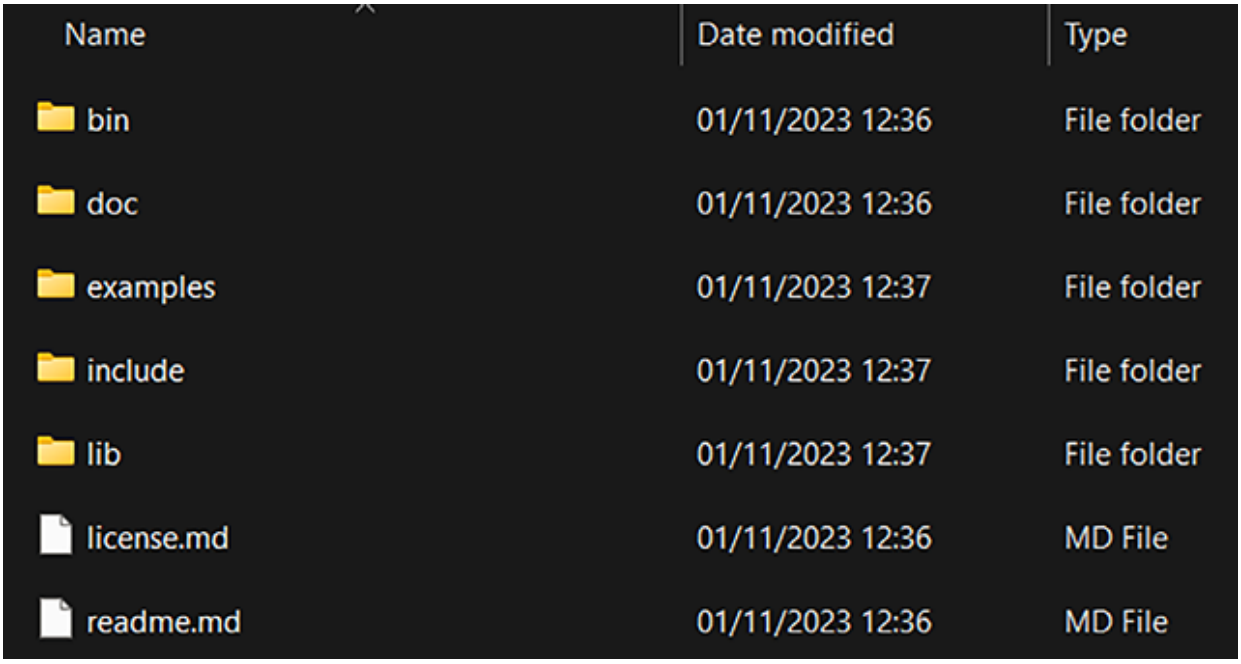


Figure 1.8: SFML folder contents

Copy the entire contents of this folder and paste all the files and folders into the SFML folder that you created in step 3. For the rest of this book, I will refer to this folder simply as "your SFML folder".

Now, we are ready to start using C++ and SFML in Visual Studio.

Creating a new project in Visual Studio 2022

As setting up a project is a fiddly process, we will go through it step by step so that we can start getting used to it:

1. Start Visual Studio in the same way you start any app: by clicking on its icon. The default installation options will have placed a Visual Studio

2022 icon in the Windows Start menu. You will see the following window:

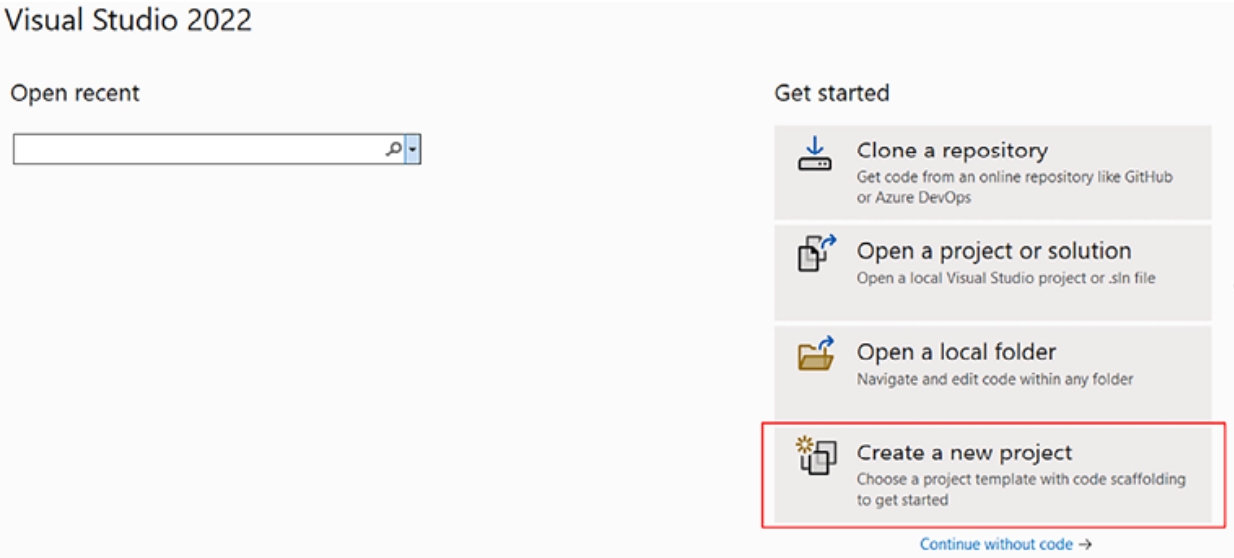

Visual Studio 2022
Open recent
Get started
Clone a repository
Get code from an online repository like GitHub or Azure DevOps
Open a project or solution
Open a local Visual Studio project or .sln file
Open a local folder
Navigate and edit code within any folder
Create a new project
Choose a project template with code scaffolding to get started
Continue without code →

Figure 1.9: Starting a new project in VS 2022

2. Click on the **Create a new project** button, as highlighted in the preceding screenshot. You will see the **Create a new project** window, as shown in the following screenshot:

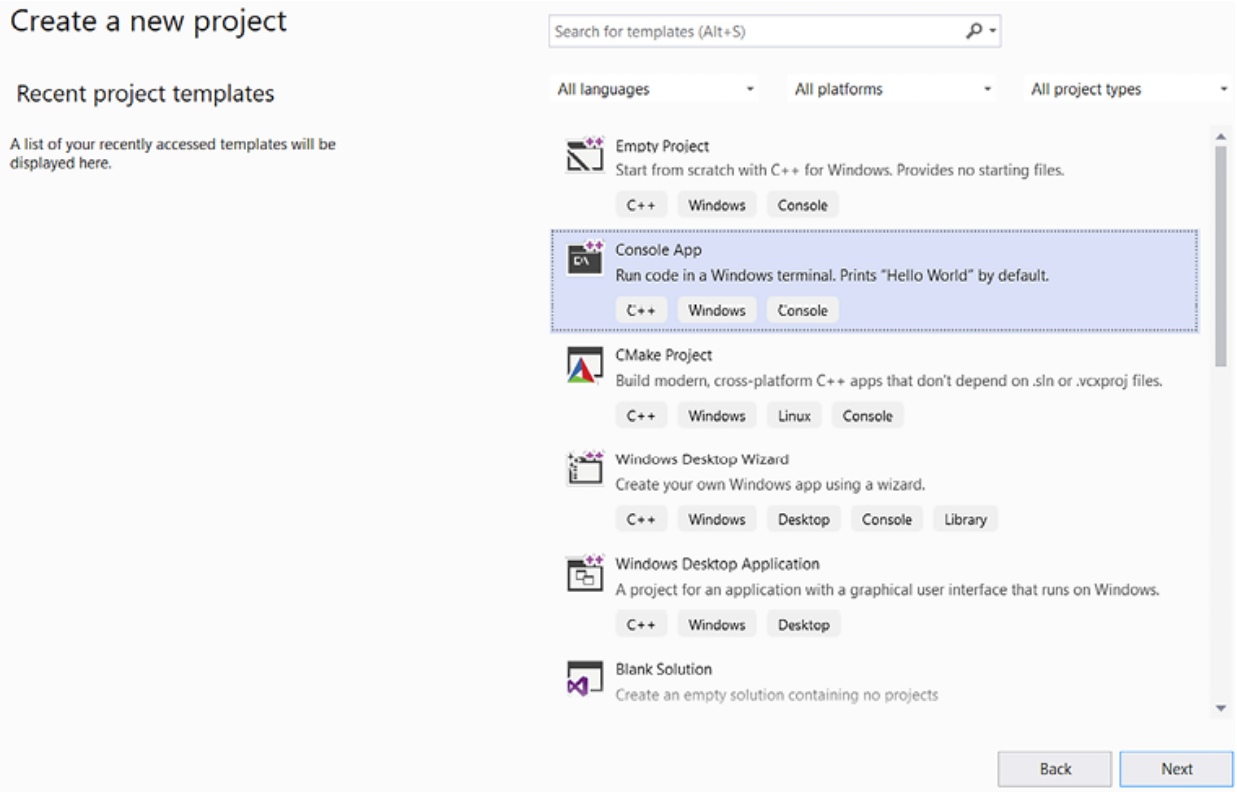

Create a new project
Recent project templates
A list of your recently accessed templates will be displayed here.
Search for templates (Alt+S)
All languages
All platforms
All project types
Empty Project
Start from scratch with C++ for Windows. Provides no starting files.
C++ Windows Console
Console App
Run code in a Windows terminal. Prints "Hello World" by default.
C++ Windows Console
CMake Project
Build modern, cross-platform C++ apps that don't depend on .sln or .vcxproj files.
C++ Windows Linux Console
Windows Desktop Wizard
Create your own Windows app using a wizard.
C++ Windows Desktop Console Library
Windows Desktop Application
A project for an application with a graphical user interface that runs on Windows.
C++ Windows Desktop
Blank Solution
Create an empty solution containing no projects
Back
Next

Figure 1.10: Create a new project screen

3. In the **Create a new project** window, we need to choose the type of project we will be creating. We will be creating a console application that has no Windows-related things like menus, selection boxes, or other Windows paraphernalia, so select **Console App**, as highlighted in the preceding screenshot, and click the **Next** button. You will then see the **Configure your new project** window. The following screenshot shows the **Configure your new project** window after the next three steps have been completed:

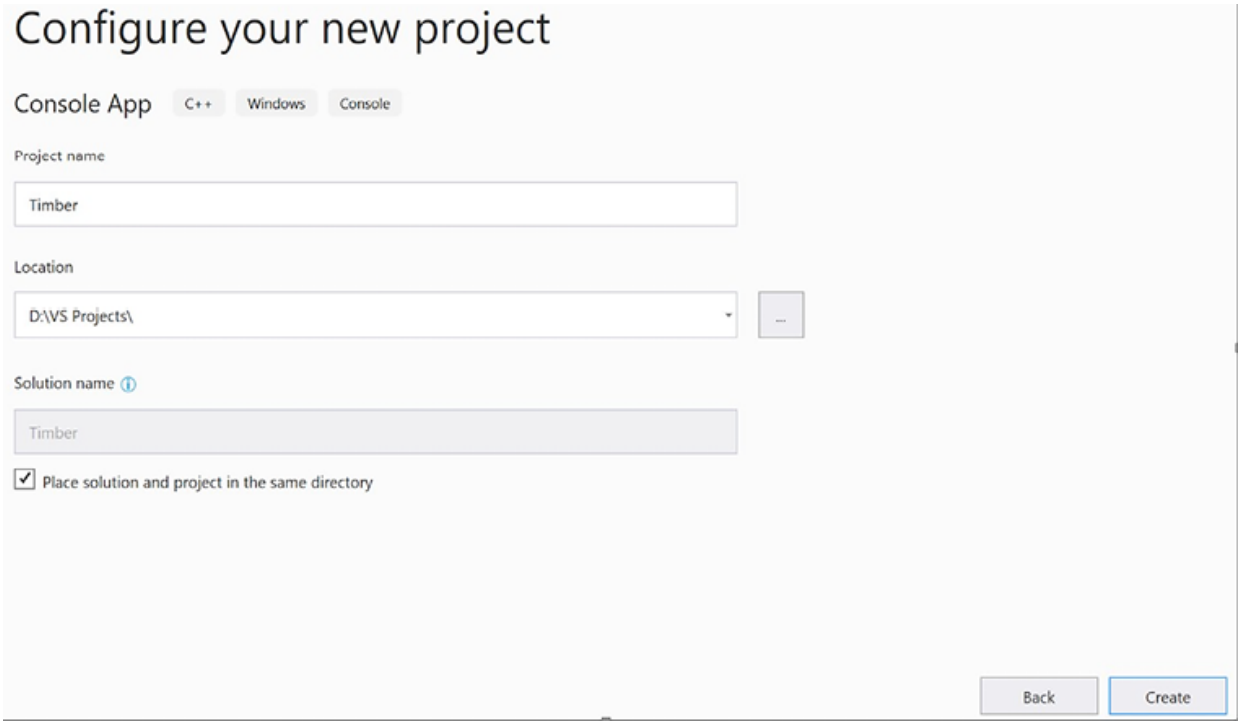


Figure 1.11: Configuring your new project

4. In the **Configure your new project** window, type `Timber` in the **Project name** field. Note that this causes Visual Studio to automatically configure the **Solution name** field to the same name.
5. In the **Location** field, browse to the `VS Projects` folder that we created in the previous tutorial. This will be the location where all our project files will be kept.
6. Check the option to place the solution and project in the same directory.
7. Note that the preceding screenshot shows what the window looks like when the previous three steps have been completed. When you have completed these steps, click **Create**. The project will be generated, including some C++ code. The following screenshot shows where we will be working throughout this book:

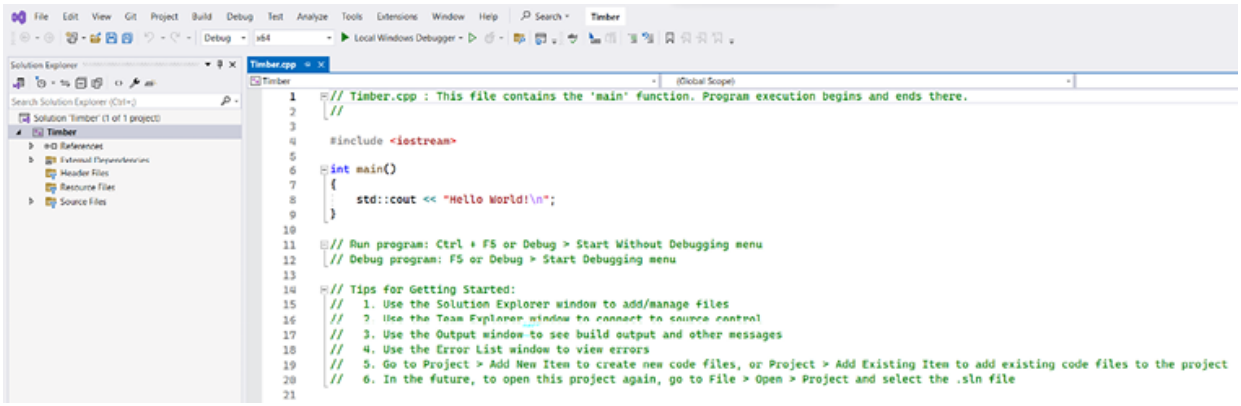

// Timber.cpp : This file contains the 'main' function. Program execution begins and ends there.
//
#include <iostream>
int main()
{
std::cout << "Hello World!\n";
}
// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu
// Tips for Getting Started:
// 1. Use the Solution Explorer window to add/manage files
// 3. Use the Output window to see build output and other messages
// 4. Use the Error List window to view errors
// 5. Go to Project > Add New Item to create new code files, or Project > Add Existing Item to add existing code files to the project
// 6. In the future, to open this project again, go to File > Open > Project and select the .sln file

Figure 1.12: Visual Studio code editor

8. We will now configure the project to use the SFML files that we put in the `SFML` folder. From the main menu, select **Project | Timber properties…**. You will see the following window:

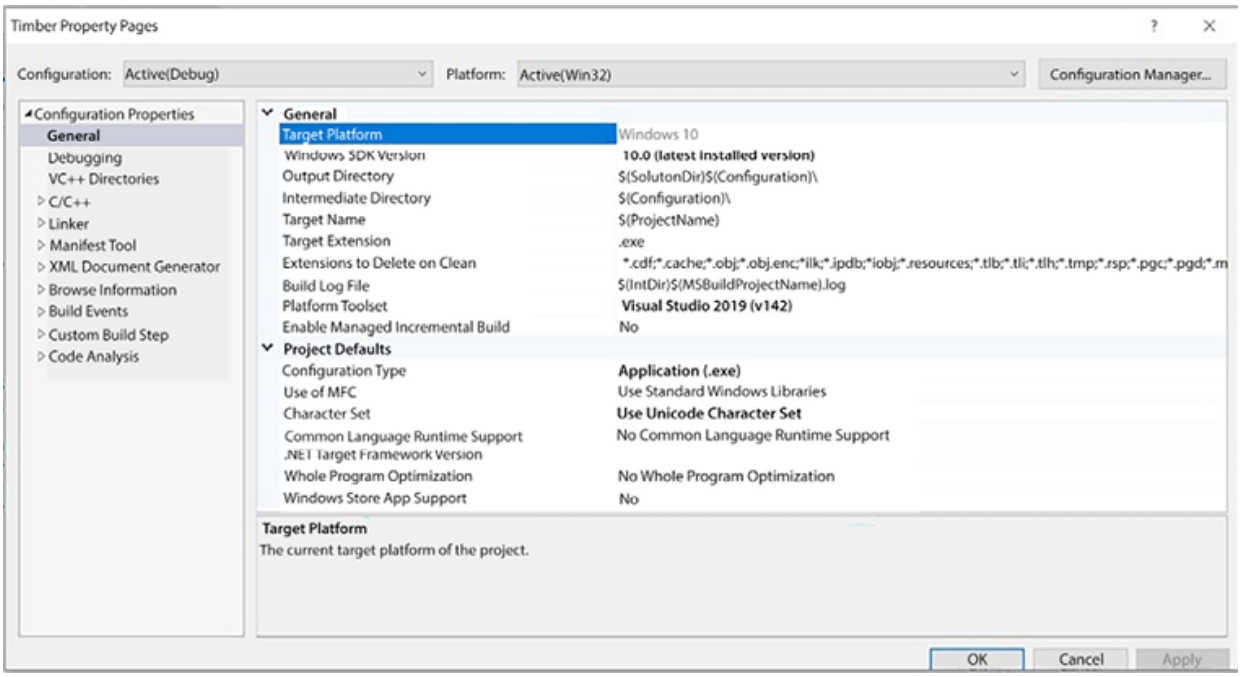


Figure 1.13: Timber Property page

In the preceding screenshot, the **OK**, **Cancel**, and **Apply** buttons are not fully formed. This is likely a glitch with Visual Studio not handling my screen resolution correctly. Yours will hopefully be fully

formed. Whether your buttons appear like mine do or not, continuing with the tutorial will be the same.

Next, we will begin to configure the project properties. As these steps are quite intricate, I will cover them in a new list of steps.

Configuring the project properties

At this stage, you should have the **Timber Property Pages** window open, as shown in the preceding screenshot at the end of the previous section. Now, we will begin to configure some properties while using the following annotated screenshot for guidance:

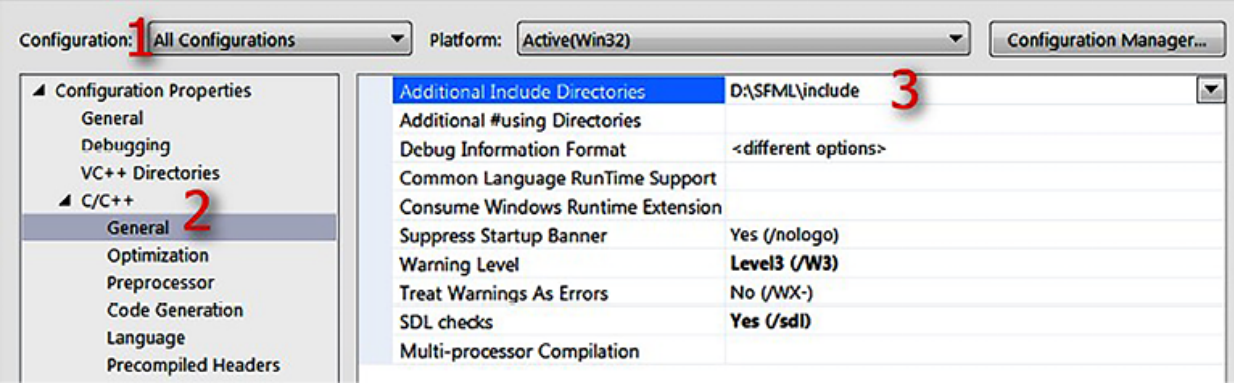


Figure 1.14: Configuring the project properties

We will add some intricate and important project settings in this section. This is the laborious part, but we will only need to do this once per project and it will get easier and faster each time you do it. What we need to do is tell Visual Studio where to find a special type of code file from SFML. The special type of file I am referring to is a header file. Header files are the files that define the format of the SFML code so that when we use the SFML code, the compiler knows how to handle it. Note that the header files are distinct from the main source code files, and they are contained in files with

the `.hpp` file extension. All this will become clearer when we eventually start adding our own header files in the second project. In addition, we need to tell Visual Studio where it can find the SFML library files. To achieve these things, on the **Timber Property Pages** window, perform the following three steps, which are numbered in the preceding screenshot:

1. First (**1**), select **All Configurations** from the **Configuration** dropdown and check that **Win32** is selected in the Platform dropdown to the right.
2. Second (**2**), select **C/C++** then **General** from the left-hand menu.
3. Third (**3**), locate the **Additional Include Directories** edit box and type the drive letter where your `SFML` folder is located, followed by `\SFML\include`. The full path to type, if you located your `SFML` folder on your D drive, is as shown in the preceding screenshot – that is, `D:\SFML\include`. Vary your path if you put SFML on a different drive.
4. Click **Apply** to save your configurations so far.
5. Now, still in the same window, perform these steps, which refer to the following annotated screenshot. First (**1**), select **Linker** and then **General**.
6. Now, find the **Additional Library Directories** edit box (**2**) and type the drive letter where your `SFML` folder is, followed by `\SFML\lib`. So, the full path to type if you located your `SFML` folder on your D drive is, as also shown in the following screenshot, `D:\SFML\lib`. Vary your path if you put SFML on a different drive:

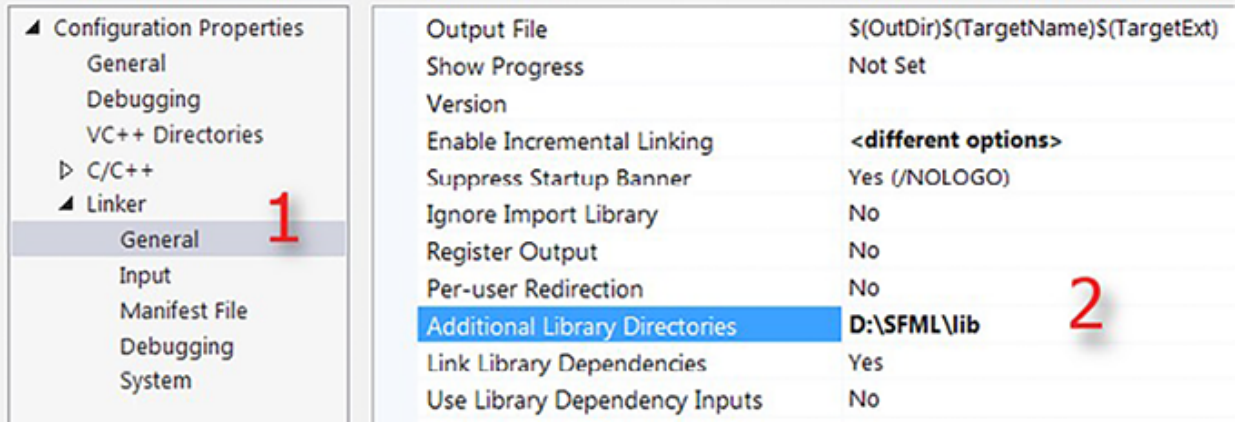


Figure 1.15: Additional Library Directories

7. Click **Apply** to save your configurations so far.
8. Finally for this stage, still in the same window, perform these steps,which refers to the following annotated screenshot. Switch the **Configuration** dropdown (**1**) to **Debug** as we will be running and testing our games in Debugging mode.

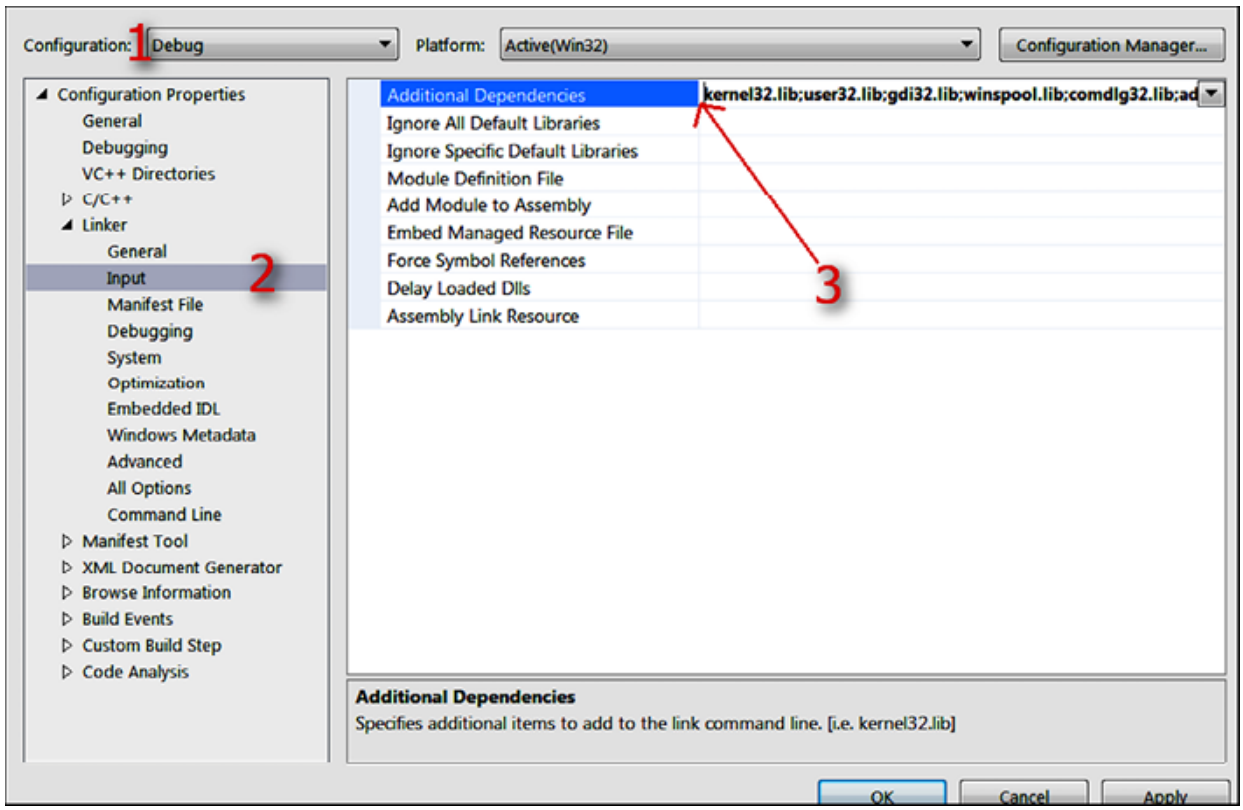


Figure 1.16: Linker input configuration

9. Select **Linker** and then **Input** (2).
10. Find the **Additional Dependencies** edit box (**3**) and click on it at the far left-hand side. Now, copy and paste/type the following: `sfml-graphics-d.lib;sfml-window-d.lib;sfml-system-d.lib;sfml-network-d.lib;sfml-audio-d.lib`; at the indicated place. Be extra careful to place the cursor exactly in the right place and not overwrite any of the text that is already there.
11. Click **OK**.
12. Click **Apply** and then **OK**.

Phew; that's it! We have successfully configured Visual Studio and can move on to planning the Timber!!! project.

Planning Timber!!!

Whenever you make a game, it is always best to start with a pencil and paper. If you don't know exactly how your game is going to work on the screen, how can you possibly make it work in code?

At this point, if you haven't already, I suggest you go and watch a video of Timberman in action so that you can see what we are aiming for. If you feel your budget can stretch to it, then grab a copy and give it a play. It is often on sale for under $1 on Steam: http://store.steampowered.com/app/398710/.

The features and objects of a game that define the gameplay are known as the **mechanics**. The basic mechanics of the game are as follows:

- Time is always running out.
- You can get more time by chopping the tree.

- Chopping the tree causes the branches to fall.
- The player must avoid the falling branches.
- Repeat until time runs out or the player is squished by a branch.

Expecting you to plan the C++ code at this stage is obviously a bit silly. This is, of course, the first chapter of a C++ beginner's guide. We can, however, look at all the assets we will use and an overview of what we will need to make our C++ code do.

Look at this annotated screenshot of the game:

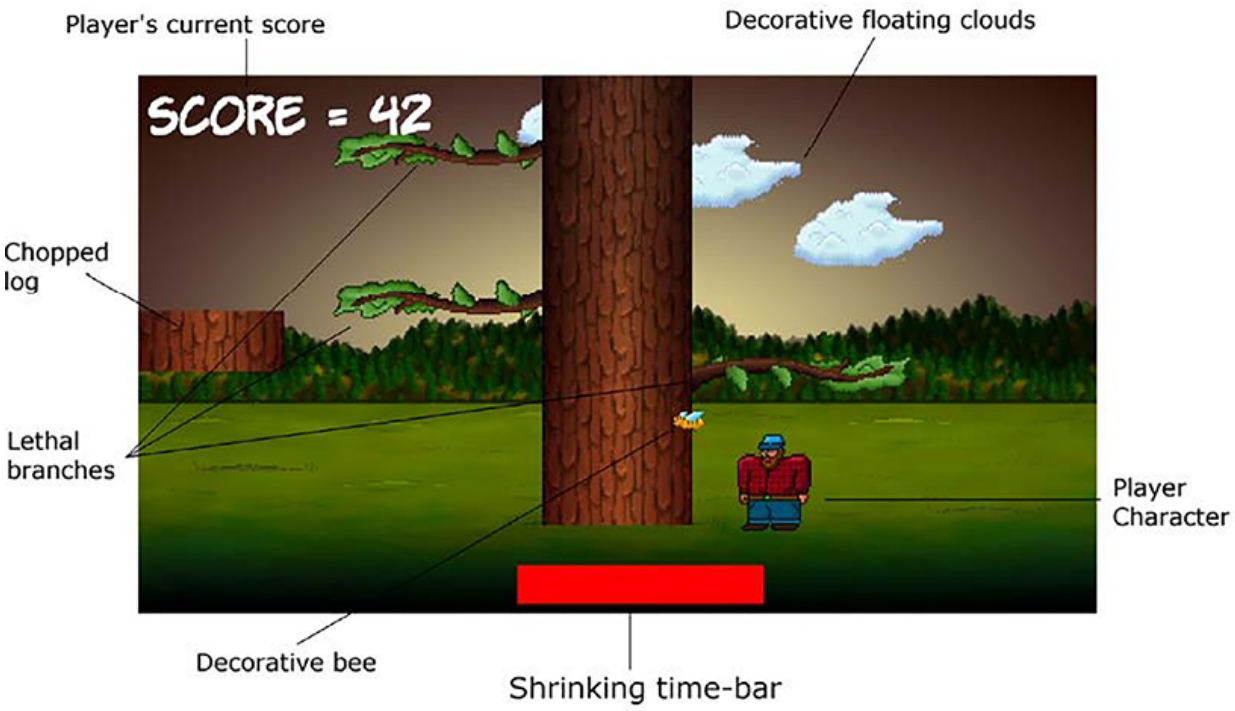


Figure 1.17: Screenshot of the Timber game

You can see that we have the following features:

- **The player's score**: Each time the player chops a log, they will get one point. They can chop a log with either the left or the right arrow (cursor) key.

- **Player character**: Each time the player chops, they will move to/stay on the same side of the tree relative to the cursor key they use. Therefore, the player must be careful which side they choose to chop on.
- When the player chops, a simple axe graphic will appear in the player character's hands.
- **Shrinking time-bar**: Each time the player chops, a small amount of time will be added to the ever-shrinking time-bar.
- **The lethal branches**: The faster the player chops, the more time they will win, but also the faster the branches will move down the tree and therefore the more likely they are to get squished. The branches spawn randomly at the top of the tree and move down with each chop.
- When the player gets squished – and they will get squished quite regularly – a gravestone graphic will appear.
- **The chopped log**: When the player chops, a chopped log graphic will whiz off, away from the player.
- **Just for decoration**: There are three floating clouds that will drift at random heights and speeds, as well as a bee that does nothing but fly around.
- **The background**: All this takes place on a pretty background.

So, in a nutshell, the player must frantically chop to gain points and avoid running out of time. As a slightly perverse but fun consequence, the faster they chop, the more likely their squishy demise.

We now know what the game looks like, how it is played, and the motivation behind the game mechanics. Now, we can go ahead and start building it. Follow these steps:

1. Now, we need to copy the SFML `.dll` files into the main project directory. My main project directory is `D:\VS Projects\Timber`. It was created by Visual Studio in the previous tutorial. If you put your `VS Projects` folder somewhere else, then perform this step there instead. The files we need to copy into the project folder are in your `SFML\bin` folder. Open a window for each of the two locations and highlight all the files in the `SFML\bin` folder, as shown in the following screenshot:

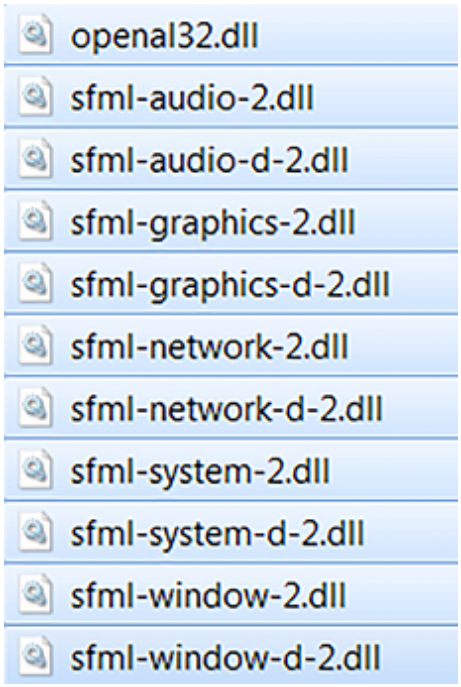


Figure 1.18: Selecting all the files you need

2. Now, copy and paste the highlighted files into the project folder – that is `D:\VS Projects\Timber`.
3. The project is now set up and ready to go. You will be able to see the following screen. I have annotated this screenshot so you can start familiarizing yourself with Visual Studio. We will revisit all these areas, and others, soon:

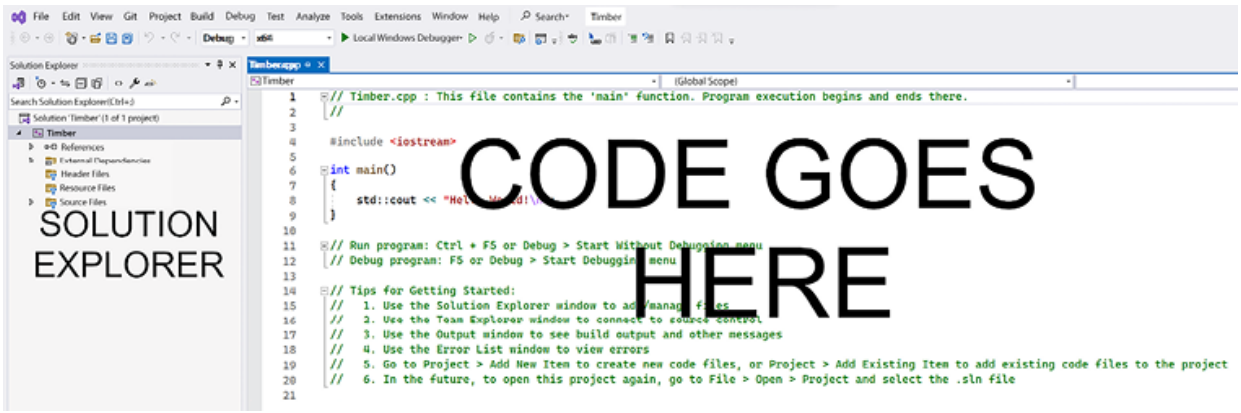


Figure 1.19: Where to type the code

Your layout might look slightly different from what's shown in the preceding screenshot because the windows of Visual Studio, like most applications, are customizable. Take the time to locate the **Solution Explorer** window and adjust it to make its content nice and clear, as shown in the previous screenshot.

We will be back here soon to start coding. But first, we will explore the project assets we will be using.

The project assets

Assets are anything you need to make your game. In our case, these assets include the following:

- A **font** for drawing the text on the screen
- Some **sound effects** for different actions, such as chopping, dying, and running out of time
- Some **graphics**, known as **textures**, for the character, background, branches, and other game objects

All the graphics and sounds that are required for this game are included in the download bundle for this book. They can be found in the `Chapter 1/graphics` and `Chapter 1/sound` folders as appropriate.

The font that is required has not been supplied. This is because I wanted to avoid any possible ambiguity regarding the license. This will not cause a problem, though, as I will show you exactly where and how to choose and download fonts for yourself.

Making your own sound FX

Sound effects (FX) can be downloaded for free from sites such as **Freesound** (www.freesound.org) but, often, the license won't allow you to use them if you are selling your game. Another option is to use an open-source software called **BFXR** from www.bfxr.net, which can help you generate lots of different sound FX that are yours to keep and do with as you like.

Adding the assets to the project

Once you have decided which assets you will use, it is time to add them to the project. The following instructions will assume you are using all the assets that are supplied in this book's download bundle. Where you are using your own, simply replace the appropriate sound or graphic file with your own, using the same filename:

1. Browse to the project folder – that is, `D:\VS Projects\Timber`.
2. Create three new folders within this folder and name them `graphics`, `sound`, and `fonts`.

3. From the download bundle, copy the entire contents of `Chapter 1/graphics` into the `D:\VS Projects\Timber\graphics` folder.
4. From the download bundle, copy the entire contents of `Chapter 1/sound` into the `D:\VS Projects\Timber\sound` folder.
5. Now, visit hhttp://www.1001freefonts.com/komika_poster.font in your web browser and download the **Komika Poster** font.
6. Extract the contents of the zipped download and add the `KOMIKAP_.ttf` file to the `D:\VS Projects\Timber\fonts` folder.

Let's look at these assets – especially the graphics – so that we can visualize what is happening when we use them in our C++ code.

Exploring the assets

The graphical assets make up the parts of the scene that is our game. If you look at the graphical assets, it should be clear where in our game they will be used:

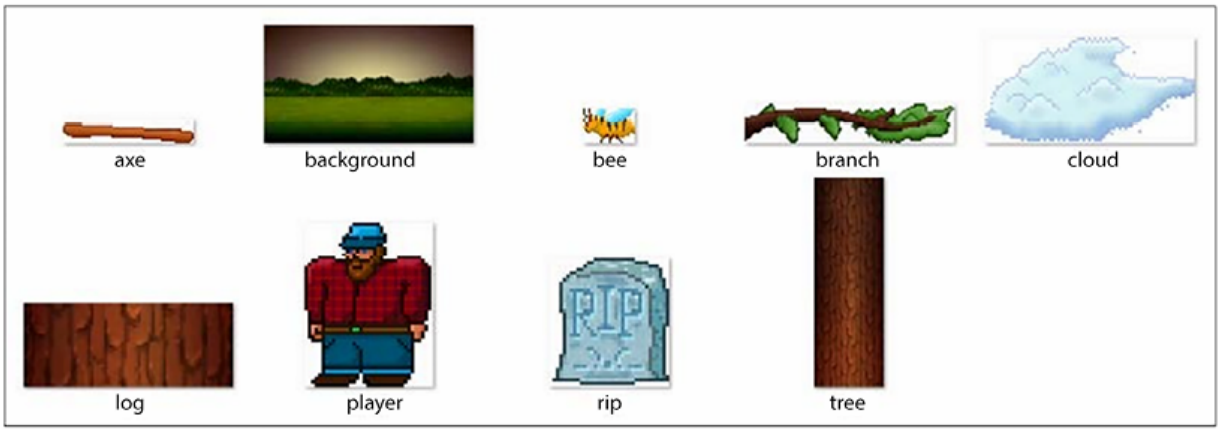


Figure 1.20: The assets

The sound files are all in `.wav` format. These files contain the sound effects that we will play at certain events throughout the game. They were all generated using **BFXR** and are as follows:

- `chop.wav`: A sound that is a bit like an axe chopping a tree
- `death.wav`: A sound a bit like a retro "losing" sound
- `out_of_time.wav`: A sound that plays when the player loses by running out of time, as opposed to being squashed

We have seen all the assets, including the graphics, so now we will have a short discussion related to the resolution of the screen and how we position the graphics on it.

Understanding screen and internal coordinates

Before we move on to the actual C++ coding, let's talk a little about coordinates. All the images that we see on our monitors are made from pixels. Pixels are tiny dots of light that combine to make the images we see on the screen.

There are many different resolutions of a monitor but, as an example, consider that a typical monitor might have 1,920 pixels horizontally and 1,080 pixels vertically.

The pixels are numbered, starting from the top left of the screen. As you can see from the following diagram, our 1,920 x 1,080 example is numbered from 0 through to 1,919 on the horizontal (x) axis and 0 through 1,079 on the vertical (y) axis:

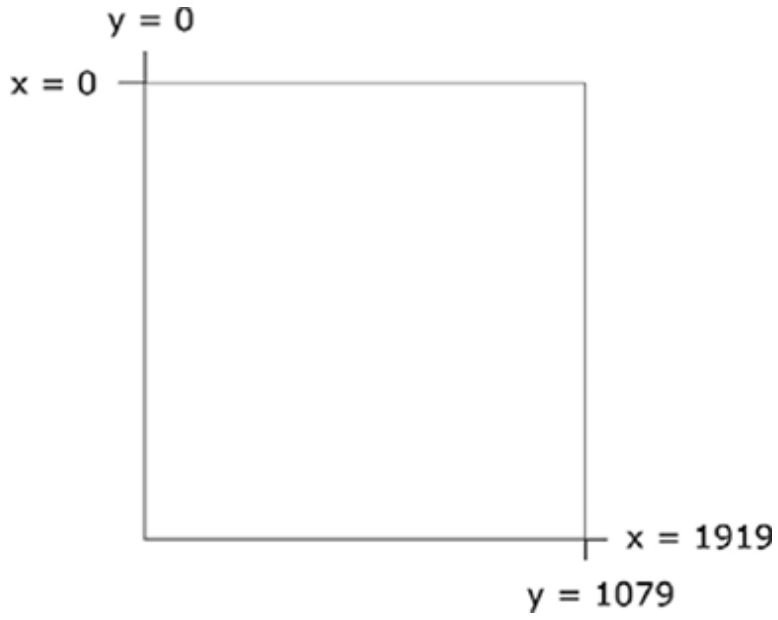


Figure 1.21: Screen and internal coordinates

A specific and exact screen location can therefore be identified by an x and y coordinate. We create our games by drawing the game objects such as the background, characters, bullets, and text to specific locations on the screen.

These locations are identified by the coordinates of the pixels. Take a look at the following hypothetical example of how we might draw at the approximately central coordinates of the screen. In the case of a 1,920 x 1080 screen, this would be at the 960, 540 position:

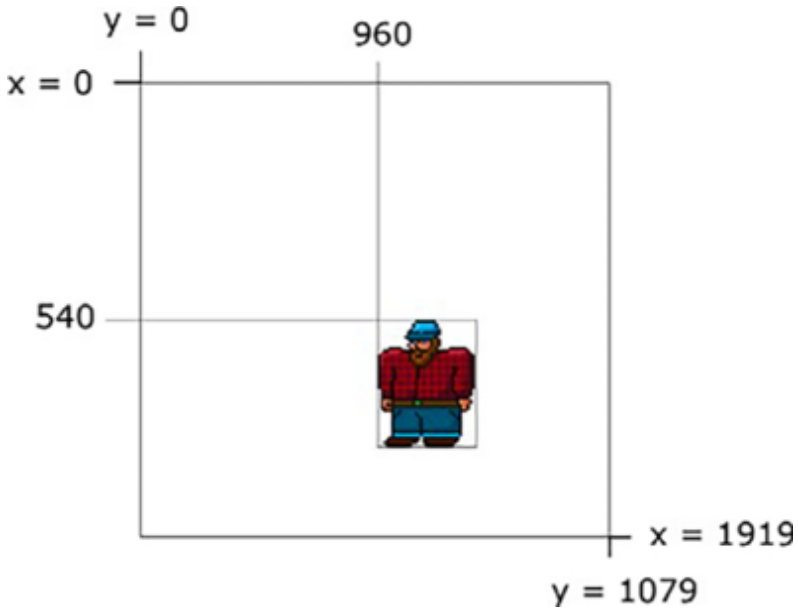


Figure 1.22: Drawing central coordinates

In addition to the screen coordinates, our game objects will each have their own similar coordinate system as well. Like the screen coordinate system, their **internal** or **local** coordinates start at 0,0 in the top-left corner.

In the previous image, we can see that 0,0 of the character is drawn at 960, 540 of the screen. A visual 2D game object, such as a character or perhaps a zombie, is called a **Sprite**. A sprite is typically made from an image file. All sprites have what is known as an origin.

If we draw a sprite to a specific location on the screen, it is the origin that will be located at this specific location. The 0,0 coordinates of the sprite are its origin. The following image demonstrates this:

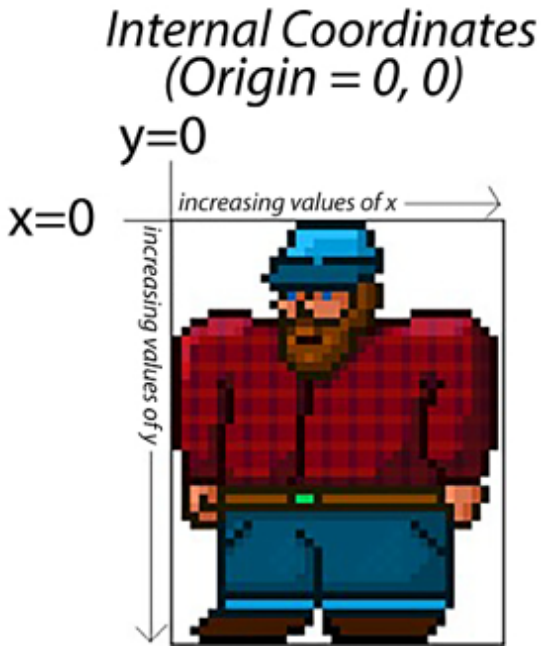


Figure 1.23: Illustration of a sprite with its origin

Therefore, in the image showing the character drawn to the screen, although we drew the image at the central position (960, 540), it appears off to the right and down.

This is important to know as it will help us understand the coordinates we use to draw all the graphics.

Note that, in the real world, gamers have a huge variety of screen resolutions, and our games will need to work with as many of them as possible. In the third project, we will see how we can make our games dynamically adapt to almost any resolution. In this first project, we will need to assume that the screen resolution is 1,920 x 1,080 or higher.

Now, we can write our first piece of C++ code and see it in action.

Getting started with coding the game

Open Visual Studio if it isn't already open. Open the **Timber** project by left-clicking it from the **Recent** list on the main Visual Studio window.

Find the **Solution Explorer** window on the right-hand side. Locate the `Timber.cpp` file under the `Source Files` folder. The `.cpp` stands for C plus plus.

Delete the entire contents of the code window and add the following code so that you have the same code yourself. You can do so in the same way that you would with any text editor or word processor; you could even copy and paste it if you prefer. After you have made the edits, we can talk about it:

```
// This is where our game starts from int main()
{
return 0;
}
```

This simple C++ program is a good place to start. Let's go through it line by line.

Making code clearer with comments

The first line of code is as follows:

```
// This is where our game starts from
```

Any line of code that starts with two forward slashes (`//`) is a comment and is ignored by the compiler. As such, this line of code does nothing. It is used to leave in any information that we might find useful when we come back to the code at a later date. The comment ends at the end of the line, so anything on the next line is not part of the comment. There is another type of

comment called a **multi-line** or **c-style** comment, which can be used to leave comments that take up more than a single line. We will see some of them later in this chapter. Throughout this book, I will leave hundreds of comments to help add context and further explain the code.

The main function

The next line we see in our code is as follows:

```
int main()
```

`int` is what is known as a **type**. C++ has many types, and they represent different *types* of data. An `int` is an **integer** or whole number. Hold that thought and we will come back to it in a minute.

The `main()` part is the name of the section of code that follows. The section of code is marked out between the opening curly brace (`{`) and the next closing curly brace (`}`).

So, everything in between these curly braces `{...}` is a part of `main`. We call a section of code like this a **function**.

Every C++ program has a `main` function and it is the place where the **execution** (running) of the entire program will start. As we progress through this book, eventually, our games will have many code files. However, there will only ever be one `main` function, and no matter what code we write, our game will always begin execution from the first line of code that's inside the opening curly brace of the `main` function.

For now, don't worry about the strange brackets that follow the function name `()`. We will discuss them further in *Chapter 4*, *Loops, Arrays, Switch,*

Enumerations, and Functions – Implementing Game Mechanics, when we get to see functions in a whole new and more interesting light.

Let's look closely at the one single line of code within our `main` function.

Presentation and syntax

Take a look at the entirety of our `main` function again:

```
int main()
{
    return 0;
}
```

We can see that, inside `main`, there is just one single line of code, `return 0;`. Before we move on to find out what this line of code does, let's look at how it is presented. This is useful because it can help us prepare to write code that is easy to read and distinguished from other parts of our code.

First, notice that `return 0;` is indented to the right by one tab. This clearly marks it out as being internal to the `main` function. As our code grows in length, we will see that indenting our code and leaving white space will be essential to maintaining readability.

Next, notice the punctuation at the end of the line. A semicolon (`;`) tells the compiler that it is the end of the instruction and that whatever follows it is a new instruction. We call an instruction that's been terminated by a semicolon a **statement**.

Note that the compiler doesn't care whether you leave a new line or even a space between the semicolon and the next statement. However, not starting a new line for each statement will lead to hard-to-read code, and missing the

semicolon altogether will result in a syntax error and the game will not compile and run.

A section of code together, often denoted by its indentation with the rest of the section, is called a **block**.

Now that you're comfortable with the idea of the `main` function, indenting your code to keep it tidy, and putting a semicolon on the end of each statement, we can move on to finding out exactly what the `return 0;` statement does.

Returning values from a function

Actually, `return 0;` does almost nothing in the context of our game. However, the concept is an important one. When we use the `return` keyword, either on its own or followed by a value, it is an instruction for the program execution to jump/move back to the code that got the function started in the first place.

Often, the code that got the function started will be yet another function somewhere else in our code. In this case, however, it is the operating system that started the `main` function. So, when `return 0;` is executed, the `main` function exits and the entire program ends.

Since we have a `0` after the `return` keyword, that value is also sent to the operating system. We could change the value of `0` to something else and that value would be sent back instead.

In programming speak, we say that the code that starts a function **calls** the function and that the function **returns** the value.

You don't need to fully grasp all this function information just yet. It is just useful to introduce it here. We will go into the full details of functions during

this first project. There's one last thing on functions that I will cover before we move on. Remember the `int` from `int main()`? This tells the compiler that the type of value that's returned from `main` must be an `int` (integer/whole number). We can return any value that qualifies as an `int`; perhaps 0, 1, 999, 6,358, and so on. If we try and return something that isn't an `int,` perhaps 12.76, then the code won't compile, and the game won't run.

Functions can return a big selection of different types, including types that we invent for ourselves! That type, however, must be made known to the compiler in the way we have just seen.

This little bit of background information on functions will make things smoother as we progress.

Running the game

You can even run the game at this point. Do so by clicking the **Local Windows Debugger** button in the quick-launch bar of Visual Studio. Alternatively, you can use the *F5* shortcut key:

Figure 1.24: The Local Windows Debugger button

Be sure that the version next to the **Local Windows Debugger** button is set to **x86**, as shown in the next image. This means our program will be 32-bit and match the version of SFML we downloaded.

Figure 1.25: Be sure you're running in x86

You will just get a black screen. If the black screen doesn't automatically close itself, you can tap any key to close it. This window is the C++ console, and we can use this to debug our game. We don't need to do this now. What is happening is that our program is starting, executing from the first line of `main`, which is `return 0;`, and then immediately exiting back to the operating system.

We now have the simplest program possible coded and running. We will now add some more code to open a window that the game will eventually appear in.

Opening a window using SFML

Now, let's add some more code. The code that follows will open a window using SFML that Timber!!! will eventually run in. The window will be 1,920 pixels wide by 1,080 pixels high and will be full screen (no border or title).

Enter the new code that is highlighted here to the existing code and then we will examine it. As you type (or copy and paste), try and work out what is going on:

```
// Include important libraries here
#include <SFML/Graphics.hpp>
// Make code easier to type with "using namespace" using namespa
// This is where our game starts from int main()
{
// Create a video mode object VideoMode vm(1920, 1080);
// Create and open a window for the game
RenderWindow window(vm, "Timber!!!", Style::Fullscreen);
return 0;
}
```

Now we will go through that code a bit at a time to understand it.

Including SFML features

The first thing we will notice in our new code is the `#include` directive.

The `#include` **directive** directs Visual Studio to *include*, or add, the contents of another file before compiling. The effect of this is that some other code, which we have not written ourselves, will be a part of our program when we run it. The process of adding code from other files into our code is called **preprocessing** and, perhaps unsurprisingly, is performed by something called a **preprocessor**. The `.hpp` file extension means it is a **header** file.

Therefore, `#include <SFML/Graphics.hpp>` tells the preprocessor to include the contents of the `Graphics.hpp` file that is contained within the folder named `SFML`. It is the same folder that we created while setting up the project.

This line adds code from the file, which gives us access to some of the features of SFML. Exactly how it achieves this will become clearer when we start writing our own separate code files and using `#include` to use them.

The most common files that we will be including throughout this book are the SFML header files that give us access to all the cool game-coding features. We will also use `#include` to access the **C++ Standard Library** header files. These header files give us access to core features of the C++ language itself.

What matters for now is that we have a whole bunch of new functionalities that have been provided by SFML available to use if we add that single line of code.

The next new line is `using namespace sf;`. We will come back to what this line does soon.

OOP, classes, and objects

We will fully discuss OOP, classes, and objects as we proceed through this book. What follows is a brief introduction so that we can understand what is happening so far.

We already know that OOP stands for object-oriented programming. OOP is a programming paradigm – that is, a way of coding. OOP is generally accepted throughout the world of programming in most languages as the best, if not the only, professional way to write code. Notice I said most; there are exceptions.

OOP introduces a lot of coding concepts, but fundamental to them all are **classes** and **objects**. When we write code, whenever possible, we want to write code that is reusable, maintainable, and secure. The way we do this is by structuring our code as a class. We will learn how to do this in *Chapter 6*, *Object-Oriented Programming – Starting the Pong Game*.

All we need to know about classes for now is that once we have coded our class, we don't just execute that code as part of our game; instead, we create usable objects from the class.

For example, if we wanted 100 zombie **NPC**s (**non-player characters**), we could carefully design and code a class called `Zombie` and then, from that single class, create as many zombie objects as we like. Each and every zombie object would have the same functionality and internal data types, but each and every zombie object would be a separate and distinct entity.

To take the hypothetical zombie example further but without showing any code for the `Zombie` class, we might create a new object based on the `Zombie` class, like this:

```
Zombie z1;
```

The `z1` object is now a fully coded and functioning `Zombie` object. We could then do this:

```
Zombie z2; Zombie z3; Zombie z4; Zombie z5;
```

We now have five separate `Zombie` instances, but they are all based on one carefully coded class. Let's take things one step further before we get back to the code we have just written. Our zombies can contain both behavior (defined by functions) as well as data, which might represent things such as the zombie's health, speed, location, or direction of travel. As an example, we could code our `Zombie` class to enable us to use our `Zombie` objects, perhaps like this:

```
z1.attack(player); z2.growl(); z3.headExplode();
```

Note again that all this zombie code is hypothetical for the moment. Don't type this code into Visual Studio – it will just produce a bunch of errors.

We would design our class so that we can use the data and behaviors in the most appropriate manner to suit our game's objectives. For example, we could design our class so that we can assign values for the data for each zombie object at the time we create it.

Let's say we need to assign a unique name and speed in meters per second at the time we create each zombie. Careful coding of the `Zombie` class could enable us to write code like this:

```
// Dave was a 100 metre Olympic champion before infection
// He moves at 10 metres per second Zombie z1("Dave", 10);
// Gill had both of her legs eaten before she was infected
// She drags along at .01 metres per second Zombie z2("Gill", .0
```

The point is that classes are almost infinitely flexible, and once we have coded the class, we can go about using them by creating an object/instance of them. It is through classes and the objects that we create from them that we will harness the power of SFML. And yes, we will also write our own classes, including a `Zombie` class.

Let's get back to the real code we just wrote.

Using namespace sf

Before we move on and look more closely at `VideoMode` and `RenderWindow`, which, as you have probably guessed by now, are classes provided by SFML, we will learn what the `using namespace sf;` line of code does.

When we create a class, we do so in a **namespace**. We do this to distinguish our classes from those that others have written. Consider the `VideoMode` class.

It is entirely possible that, in an environment such as Windows, somebody has already written a class called `VideoMode`. By using a namespace, we and the SFML programmers can make sure that the names of classes never clash.

The full way of using the `VideoMode` class is like this:

```
sf::VideoMode...
```

`using namespace sf;` enables us to omit the `sf::` prefix from everywhere in our code. Without it, there would be over 100 instances of `sf::` in this simple game alone. It also makes our code more readable, as well as shorter.

SFML VideoMode and RenderWindow

Inside the `main` function, we now have two new comments and two new lines of executable code. The first line of executable code is this:

```
VideoMode vm(1920, 1080);
```

This code creates an object called `vm` from the class called `VideoMode` and sets up two internal values of `1920` and `1080`. These values represent the resolution of the player's screen.

The next new line of code is as follows:

```
RenderWindow window(vm, "Timber!!!", Style::Fullscreen);
```

In the previous line of code, we are creating a new object called `window` from the SFML-provided class called `RenderWindow`. Furthermore, we are setting up some values inside our window object.

Firstly, the `vm` object is used to initialize part of `window`. At first, this might seem confusing. Remember, however, that a class can be as varied and flexible as its creator wants to make it. And yes, some classes can contain instances of other classes.

It is not necessary to fully understand how this works at this point if you appreciate the concept. We code a class and then make usable objects from that class – a bit like an architect might draw a blueprint. You certainly can't move all your furniture, kids, and the dog into the blueprint, but you could build a house (or many houses) from the blueprint. In this analogy, a class is like a blueprint and an object is like a house.

Next, we use the `"Timber!!!"` value to give the window a name. Then, we use the predefined `Style::FullScreen` value to make our `window` object full-screen.

`Style::FullScreen` is a value that's defined in SFML. It is useful because we don't need to remember the integer number the internal code uses to represent a full screen. The coding term for this type of value is `constant`. Constants and their close C++ relatives, **variables**, are covered in the next chapter.

Let's look at our window object in action.

Running the game

You can run the game again at this point. You will see a bigger black screen flash on and then disappear. This is the 1920 x 1080 full-screen window that we just coded. Unfortunately, what is still happening is that our program is starting, executing from the first line of `main`, creating the cool new game window, then coming to `return 0;` and immediately exiting back to the operating system.

Next, we will add some code that will form the basic structure of every game in this book. This is known as the game loop.

The game loop

These are some things that we need our program to do that we will achieve in this section. We need a way to stay in the program until the player wants to quit. At the same time, we should clearly mark out where the different parts of our code will go as we progress with Timber!!!. Furthermore, if we are going to stop our game from exiting, we had better provide a way for the player to exit when they are ready; otherwise, the game will go on forever!

Add the following highlighted code to the existing code and then we will go through it and discuss it all:

```
int main()
{
// Create a video mode object VideoMode vm(1920, 1080);
// Create and open a window for the game
RenderWindow window(vm,"Timber!!!", Style::Fullscreen);
while (window.isOpen())
{
/*
****************************************** Handle the players inpu
******************************************
*/
if (Keyboard::isKeyPressed(Keyboard::Escape))
{
window.close();
}
/*
****************************************** Update the scene
******************************************
*/
/*
****************************************** Draw the scene
******************************************
*/
// Clear everything from the last frame window.clear();
// Draw our game scene here
// Show everything we just drew window.display();
```

```
}
return 0;
}
```

Next, we will go through and explain the code we have just added.

while loops

The very first thing we saw in the new code is as follows:

```
while (window.isOpen())
{
```

The very last thing we saw in the new code is a closing `}`. We have created a **while** loop. Everything between the opening (`{`) and closing (`}`) brackets of the `while` loop will continue to execute over and over, potentially forever.

Look closely between the parentheses `(...)` of the `while` loop, as highlighted here:

```
while (window.isOpen())
```

The full explanation of this code will have to wait until we discuss loops and conditions in *Chapter 4*, *Loops, Arrays, Switch, Enumerations, and Functions – Implementing Game Mechanics*. What is important for now is that when the `window` object we coded previously is set to closed, the execution of the code will break out of the `while` loop and move on to the next statement. Exactly how a window is closed is covered soon.

The next statement is, of course, `return 0;`, which ends our game.

We now know that our `while` loop will whiz round and round, repeatedly executing the code within it, until our window object is set to closed.

C-style code comments

Just inside the `while` loop, we can see what, at first glance, might look a bit like ASCII art:

```
/*
**************************************** Handle the player's inp
****************************************
*/
```

ASCII art is a niche but fun way of creating images with computer text. You can read more about it here: https://en.wikipedia.org/wiki/ASCII_art.

The previous code is simply another type of comment. This type of comment is known as a **C-style** comment. The comment begins with (`/*`) and ends with (`*/`). Anything in between is just for information and is not compiled. I have used this slightly elaborate text to make it absolutely clear what we will be doing in each part of the code file. And, of course, you can now work out that any code that follows will be related to handling the player's input.

Skip over a few lines of code and you will see that we have another C-style comment, announcing that in that part of the code, we will be updating the scene.

If you jump to the next C-style comment, it will be clear where we will be drawing all the graphics.

Let's go into these sections in more detail.

Input, update, draw, repeat

Although this first project uses the simplest possible version of a game loop, every game will need these phases in the code. Let's go over the steps:

1. Get the player's input (if any).
2. Update the scene based on things such as artificial intelligence, physics, or the player's input.
3. Draw the current scene.
4. Repeat these steps at a fast enough rate to create an interactive, smooth, animated game world.

Now, let's look at the code that does something within the game loop.

Detecting a key press

Firstly, within the section that's identifiable by the comment with the `Handle the player's input` text, we have the following code:

```
if (Keyboard::isKeyPressed(Keyboard::Escape))
{
window.close();
}
```

This code checks whether the *Esc* key is currently being pressed. If it is, the highlighted code uses the `window` object to close itself. Now, the next time the `while` loop begins, it will see that the `window` object is closed and jump to the code immediately after the closing curly brace of the `while` loop and

the game will exit. We will discuss `if` statements more fully in *Chapter 2*, *Variables, Operators, and Decisions – Animating Sprites*

Clearing and drawing the scene

Currently, there is no code in the `Update the scene` section, so let's move on to the `Draw the scene` section. The first thing we will do is rub out the previous frame of animation using the following code:

```
window.clear();
```

What we would do now is draw every object from the game. However, we don't have any game objects yet.

The next line of code is as follows:

```
window.display();
```

When we draw all the game objects, we are drawing them to a hidden surface ready to be displayed. The `window.display()` code flips from the previously displayed surface to the newly updated (previously hidden) one. This way, the player will never see the drawing process as the surface has all the sprites added to it. It also guarantees that the scene will be complete before it is flipped. This prevents a graphical glitch known as **tearing**. This process is called **double buffering**.

Also note that all this drawing and clearing functionality is performed using our `window` object, which was created from the SFML `RenderWindow` class.

Running the game

Run the game and you will get a blank, full-screen window that remains open until you press the *Esc* key.

That is good progress. At this stage, we have an executing program that opens a window and loops around, waiting for the player to press the *Esc* key to exit. Now, we can move on to drawing the background image of the game.

Drawing the game's background

Now, we will get to see some graphics in our game. What we need to do is create a sprite. The first one we will create will be the game background. We can then draw it in between clearing the window and displaying/flipping it.

Preparing the sprite using a texture

The SFML `RenderWindow` class allowed us to create our `window` object, which took care of all the functionality that our game's window needs.

We will now look at two more SFML classes that will take care of drawing sprites to the screen. One of these classes, perhaps unsurprisingly, is called `Sprite`. The other class is called `Texture`. A texture is a graphic stored in video memory, on the **graphics processing unit** (**GPU**).

An object that's made from the `Sprite` class needs an object made from the `Texture` class to display itself as an image. Add the following highlighted code. Try and work out what is going on as well. Then, we will go through it, a line at a time:

```
int main()
{
// Create a video mode object VideoMode vm(1920, 1080);
```

```
// Create and open a window for the game
RenderWindow window(vm,"Timber!!!", Style::Fullscreen);
// Create a texture to hold a graphic on the GPU Texture texture
// Load a graphic into the texture textureBackground.loadFromFil
// Create a sprite Sprite spriteBackground;
// Attach the texture to the sprite spriteBackground.setTexture(
// Set the spriteBackground to cover the screen spriteBackground
while (window.isOpen())
{
```

A point worth noting is that this code comes before the loop because it only needs to happen once. First, we create an object called `textureBackground` from the SFML `Texture` class:

```
Texture textureBackground;
```

Once this is done, we can use the `textureBackground` object to load a graphic from our `graphics` folder into `textureBackground`, like this:

```
textureBackground.loadFromFile("graphics/background.png");
```

We only need to specify `graphics/background` as the path is relative to the Visual Studio **working directory** where we created the folder and added the image.

Next, we create an object called `spriteBackground` from the SFML `Sprite` class with this code:

```
Sprite spriteBackground;
```

Then, we can associate the `Texture` object (`backgroundTexture`) with the `Sprite` object (`backgroundSprite`), like this:

```
spriteBackground.setTexture(textureBackground);
```

Finally, we can position the `spriteBackground` object in the `window` object at the `0,0` coordinates – the top-left corner:

```
spriteBackground.setPosition(0,0);
```

Since the `background.png` graphic in the `graphics` folder is 1,920 pixels wide by 1,080 pixels high, it will neatly fill the entire screen. Just note that this previous line of code doesn't show the sprite. It just sets its position, ready for when it is shown.

The `backgroundSprite` object can now be used to display the background graphic.

Of course, you are almost certainly wondering why we had to do things in such a convoluted way. The reason is because of the way that graphics cards and OpenGL work.

Textures take up graphics memory, and this memory is a finite resource. Furthermore, the process of loading a graphic into the GPU's memory is very slow – not so slow that you can watch it happen or that you will see your PC noticeably slow down while it is happening, but slow enough that you can't do it every frame of the game loop. So, it is useful to disassociate the texture (`textureBackground`) from any code that we will manipulate during the game loop.

As you will see when we start to move our graphics, we will do so using the sprite. Any objects that are made from the `Texture` class will sit happily on the GPU, just waiting for an associated `Sprite` object to tell it where to show itself. In later projects, we will also reuse the same `Texture` object

with multiple different `Sprite` objects, which makes efficient use of GPU memory.

In summary, we can state the following:

- Textures are very slow to load onto the GPU.
- Textures are very fast to access once they are on the GPU.
- We associate a `Sprite` object with a texture.
- We manipulate the position and orientation of `Sprite` objects (usually in the `Update the scene` section).
- We draw the `Sprite` object, which, in turn, displays the `Texture` object that is associated with it (usually in the `Draw the scene` section).

So, all we need to do now is use our double buffering system, which is provided by our `window` object, to draw our new `Sprite` object (`spriteBackground`), and we should get to see our first graphics in action.

Double buffering the background sprite

Finally, we need to draw that sprite and its associated texture in the appropriate place in the game loop.

Note that when I present code that is all from the same block, I don't add the indentations because it lessens the instances of line wraps in the text of the book. The indenting is implied. Check out the code file in the download bundle to see the full use of indenting.

Add the following highlighted code:

```
/*
****************************************** Draw the scene
******************************************
*/
// Clear everything from the last run frame window.clear();
// Draw our game scene here
window.draw(spriteBackground);
// Show everything we just drew window.display();
```

The new line of code simply uses the `window` object to draw the `spriteBackground` object, in between clearing the display and showing the newly drawn scene.

We now know what a sprite is, that we can associate a texture with it and then position it on the screen, and finally, draw it. The game is ready to be run again so that we can see the results of this code.

Running the game

If we run the program now, we will see the first signs that we have a real game in progress:

Figure 1.26: Running the game

It's not going to get Game of the Year in its current state, but we are on the way at least!

Let's look at some of the things that might go wrong in this chapter and as we proceed through this book.

Handling errors

There will always be problems and errors in every project you make. This is guaranteed! The tougher the problem, the more satisfying it is when you solve it. When, after hours of struggling, a new game feature finally bursts into life, it can cause a genuine high. Without this struggle, it would somehow be less worthwhile.

At some point in this book, there will probably be some struggle. Remain calm, be confident that you will overcome it, and then get to work.

Remember that whatever your problem, it is very likely you are not the first person in the world to have had this same problem. Think of a concise sentence that describes your problem or error and then type it into **Google** or **ChatGPT**. You will be surprised at the speed and precision of solving a problem this way as, often, someone else will have already solved your problem for you.

Having said that, here are a few pointers to get you started in case you are struggling with making this first chapter work.

Configuration errors

The most likely cause of problems in this chapter will be **configuration errors**. As you probably noticed during the process of setting up Visual Studio, SFML, and the project itself, there are an awful lot of filenames, folders, and settings that need to be just right. Just one wrong setting could cause one of several errors, whose text doesn't make it clear exactly what is wrong.

If you can't get the empty project with the black screen working, it might be easier to start again. Make sure all the filenames and folders are appropriate for your specific setup and then get the simplest part of the code running. This is the part where the screen flashes black and then closes. If you can get to that stage, then configuration is probably not the issue.

Compile errors

Compile errors are probably the most common errors we will experience going forward. Check that your code is identical to mine, especially semicolons on the ends of lines and subtle changes in upper- and lowercase

for class and object names. If all else fails, open the code files in the download bundle and copy and paste it in.

While it is always possible that a code typo made it into this book, the code files were made from real working projects – they definitely work!

Link errors

Link errors are most likely caused by missing SFML `.dll` files. Did you copy all of them into the project folder?

Bugs

Bugs are what happen when your code works but not as you expect it to. Debugging can actually be fun. The more bugs you squash, the better your game and the more satisfying your day's work will be. The trick to solving bugs is to find them early! To do this, I recommend running and playing your game every time you implement something new. The sooner you find the bug, the more likely the code causing it will be fresh in your mind. In this book, we will run the code to see the results at every possible stage.

Summary

This was quite a challenging chapter. It is true that configuring an IDE to use a C++ library can be a bit awkward and long. Also, the concepts of classes and objects are well known to be slightly awkward for people who are new to coding.

Now that we are at this stage, however, we can focus on C++, SFML, and games. As we progress with this book, we will learn more and more C++, as well as implement increasingly interesting game features. As we do so, we

will take a further look at things such as functions, classes, and objects to help demystify them a little more.

We have achieved plenty in this chapter, including outlining a basic C++ program with the main function and constructing a simple game loop that listens for player input and draws a sprite (along with its associated texture) to the screen.

In the next chapter, we will learn about all the C++ we need to draw some more sprites and animate them as well.

Frequently asked questions

Here are some questions that might be on your mind:

Q) I am struggling with the content that's been presented so far. Am I cut out for programming?

A) Setting up a development environment and getting your head around OOP as a concept is probably the toughest thing you will do in this book. If your game is functioning (drawing the background), you are ready to proceed with the next chapter.

Q) All this talk of OOP, classes, and objects is too much and kind of spoiling the whole learning experience.

A) Don't worry. We will keep returning to OOP, classes, and objects constantly. In *Chapter 6*, *Object-Oriented Programming – Starting the Pong Game*, we will really begin getting to grips with the whole OOP thing. All you need to understand for now is that SFML has written a whole load of useful classes and that we get to use this code by creating usable objects from those classes. When you learn more about OOP, you will feel empowered.

Q) I really don’t get this function stuff.

A) It doesn’t matter; we will be returning to it again constantly and will learn about functions more thoroughly. You just need to know that, when a function is called, its code is executed, and when it is done (reaches a `return` statement), the program jumps back to the code that called it.

2

Variables, Operators, and Decisions: Animating Sprites

In this chapter, we will do quite a bit more drawing on the screen. We will animate some clouds that travel at a random height and a random speed across the background and a bee that does the same in the foreground. To achieve this, we will need to learn some more of the basics of C++. We will be learning how C++ stores data with variables as well as how to manipulate those variables with the C++ operators and how to make decisions that branch our code on different paths based on the value of variables. Once we have learned all this, we will be able to reuse our knowledge about the **Simple and fast Multimedia Library** (**SFML**) **Sprite** and **Texture** classes to implement our cloud and bee animations.

In summary, here is what is in store:

- Learning all about C++ variables
- Seeing how to manipulate the variables
- Adding clouds, a buzzing bee and a tree for the player to chop away at
- Random numbers
- Making decisions with `if` and `else`
- Timing
- Moving the clouds and the bee

Learning all about C++ variables

Variables are the way that our C++ games store and manipulate the values/data in our game. If we want to know how much health the player has, we need a variable. Perhaps you want to know how many zombies are left in the current wave? That is a variable as well. If you need to keep track of the name of the player who got a specific high score, you guessed it, we need a variable for that. Is the game over or still playing? Yep, that's a variable too.

Variables are named identifiers to **locations in memory**. So, we might name a variable called `numberOfZombies`, and that variable could refer to a place in memory that stores a value to represent the number of zombies that are remaining in the current wave.

The way that computer systems address locations in memory is complex. Programming languages use variables to give a human-friendly way to manage our data in that memory. Managing a complex system in a human-friendly way is really what programming languages are. What varies from language to language is how efficient and friendly they are. C++ has always been efficient and in the course of its history has become progressively more user friendly, too.

C++ was created by Bjarne Stroustrup in the early 1980s. C++ evolved from the original C language. Stroustrup developed C++ with the aim of adding object-oriented programming features to C, allowing for more efficient and manageable code. Over the years, C++ has had many revisions/improvements.

The small amount we have just mentioned about variables implies that there must be different *types* of variables. There are many types of variables in C++. Let's look at the ones we will use the most over the course of this book.

Types of variables

It would easily be possible to spend an entire chapter discussing C++ variables and types. There are already numerous books that do this, so I am not going to do so here because I am guessing you are here for the fastest path possible to building games. Therefore, what follows is a table of the most used types of variables in this book. Then, we will look at how to use each of these variable types.

Type	**Examples of values**	**Explanation**
`int`	`-42`, `0`, `1`, and `9826`.	Integer whole numbers.
`float`	`-1.26f`, `5.8999996f` and `10128.3f`.	Floating point values with precision up to 7 digits.
`double`	`925.83920655234` and `1859876.94872535`.	Floating point values with precision up to 15 digits.
`char`	`a`, `b`, `c`, `1`, `2`, and `3` (a total of 128 symbols including `?`, `~`, `#`, etc...).	Any symbol from the ASCII table (see next tip about variables).
`bool`	`True` or `false`.	`bool` stands for Boolean and can be only `true` or `false`.
`String`	`Hello Everyone! I am a String.`	Any text value from a single letter or digit up to perhaps an entire book.

Table 2.1 Types of variables

C++ is **strongly typed**. In programming languages, strong typing refers to a system in which the data type of a variable is strictly enforced, and implicit type conversions are limited. In a strongly typed language, operations between different data types often require explicit conversions or they will result in compiler errors. This strict enforcement reduces the likelihood of unexpected behaviors in our games, as the compiler or interpreter ensures that variables are used in a manner consistent with their declared types.

For these reasons, the compiler must be told what type a variable is, so that it can allocate the right amount of memory for it. Furthermore, when the compiler knows what type a variable is, it can check that it is not being used in an erroneous way. For example, you wouldn't divide a string by a `bool`. It is good practice to use the best and most appropriate type for each variable you use. In practice, however, you will often get away with promoting a variable to a more precise type. Perhaps you need a floating-point number with just five significant digits? The compiler won't complain if you store it as a `double`. However, if you try to store a `float` or a `double` in an `int`, it will change/cast the value to fit the `int`. This will also change the value that is stored. As we progress through the book, I will make it plain what the best variable type is to use in each case and we will even see a few instances where we deliberately convert/cast between variable types.

A few extra details worth noticing in the preceding table include the `f` postfix next to all the `float` values. This `f` tells the compiler that the value is type `float` not `double`. A floating-point value without the `f` prefix is assumed to be `double`. See the next tip about variables for more about this.

User-defined types

User-defined types are way more advanced than the types we have just seen. When we talk about user-defined types in C++, we are usually talking about classes or enumerations. We briefly talked about classes and their related objects in the previous chapter. Soon we will write code in a separate file, sometimes two separate files. We will then be able to declare, initialize, and use the classes that we design. We will leave how we define/create our own types until *Chapter 6*, *Object-Oriented Programming: Starting the Pong Game*. We will see enumerations in *Chapter 4*. Enumerations act as a gentle introduction to classes as they are a way for the programmer to define their own types, perhaps types of zombies, power-ups, or alien spaceships. Let's get back to the built-in basic C++ types often referred to as the **fundamental types** because they represent fundamental values like those we saw in the preceding table.

Declaring and initializing variables

So far, we know that variables are for storing the data/values that our games need to work. For example, a variable would represent the number of lives a player has or the player's name. We also know that there is a wide selection of different types of values that these variables can represent, such as `int`, `float`, `bool`, or user defined. Of course, what we haven't seen yet is how we would go about using a variable.

There are two stages for creating and preparing a new variable. The stages are called **declaration** and **initialization**. Let's look at each in turn.

Declaring variables

We can declare variables in C++ like this:

```
// What is the player's score?
int playerScore;
// What is the player's first initial
char playerInitial;
// What is the value of pi
float valuePi;
// Is the player alive or dead?
bool isAlive;
```

In the preceding code, we have declared an int called `playerScore`, a `char` called `playerInitial`, a `float` called `valuePi`, and a `bool` called `isAlive`. If you need a reminder of exactly what these different types are, check back to the previous table. What we have achieved by these declarations is that we have reserved appropriately sized places in memory to store and manipulate values of the appropriate types. We haven't got any data yet. Let's keep going and find out more.

Initializing variables

Now that we have declared the variables with meaningful names, we can initialize those same variables with appropriate values, like this:

```
playerScore = 0;
playerInitial = 'J';
valuePi = 3.141f;
isAlive = true;
```

Now, if we execute the preceding code, we have real data in the memory of the computer. In case it isn't obvious, the four preceding variables hold the values of zero, the lowercase letter `j`, the floating point number `3.141`, and the binary value `true`.

Declaring and initializing in one step

When it suits us, we can combine the declaration and initialization steps into one. If we know the values we want our variables to start with, we could code them like this next example.

```
int playerScore = 0;
char playerInitial = 'J';
float valuePi = 3.141f;
bool isAlive = true;
```

If we needed to determine the value of our variables during program execution, we would more likely code them as we did in the first examples of the variables. Both are correct to C++, but, usually, one way is more appropriate for your game.

If you want to see a complete list of C++ types, then check this web page: http://www.tutorialspoint.com/cplusplus/cpp_data_types.htm. If you want a deeper discussion on `float`, `double`, and the `f` postfix, then read this: http://www.cplusplus.com/forum/beginner/24483/. If you want to know about ASCII character codes, then there is some more information here: http://www.cplusplus.com/doc/ascii/. Note that these links are for the extra curious reader, and we have already discussed enough in order to proceed.

Constants

Sometimes we need to make sure that a value can never be changed. To achieve this, we can declare and initialize a **constant** using the `const` keyword. The value of Pi doesn't change, so it would be more correct in most cases to have used a constant variable.

```
const float PI = 3.141f;
const int NUMBER_OF_ENEMIES = 2000;
```

In the preceding code, we guarantee that the value of the `PI` variable can never change during program execution, and neither can the `NUMBER_OF_ENEMIES` variable. When declaring constants, it is common to use a different format. The format we will use in this book will be all uppercase with the words denoted by underscores instead of camel casing.

To be clear, when I say that a constant can never be changed, I mean it can't be changed by the program execution. As a programmer, you can always change the value of your constants at initialization time; you just can't write code to change them during execution.

```
//const int PLANETS_IN_SOLAR_SYSTEM = 9;
// Whoops! Pluto reclassified to dwarf planet in 2006.
const int PLANETS_IN_SOLAR_SYSTEM = 8;
```

We will see some constants in action in *Chapter 4*, *Loops, Arrays, Switch, Enumerations, and Functions: Implementing Game Mechanics*.

There is another variable initialization topic to discuss.

Uniform initialization

Uniform initialization or list initialization is a newer way to initialize variables. Uniform initialization in C++ began with the introduction of

C++11, which was a major update to the C++ programming language in 2011. Uniform initialization provides a more consistent syntax for initializing variables and our user-defined types. It allows initialization using curly braces, `{}`, just like the curly braces that wrap the `main` function. You can use uniform initialization for the variables we saw previously as follows:

```
int playerScore{0};
char playerInitial{'J'};
float valuePi{3.141f};
bool isAlive{true};
```

In the preceding code, I've replaced the assignment operator, `=`, with the uniform initialization syntax, `{}`, for each variable. This syntax is the formal standard in modern C++ and you will most often see it in modern commercial APIs. There are some advanced reasons why it is less error prone than the "traditional" method, which we will use in this book.

It's not wrong to use the traditional syntax like this:

```
int playerScore = 0;
```

Both approaches are valid and will work. I just wanted you to see the syntax you will sometimes come across when exploring C++ elsewhere. Furthermore, we will bump into this style later in *Chapter 6* when we talk about classes. Feel free to use uniform initialization throughout the book. It would be simple to modify all the code samples. My view is that the traditional syntax is more beginner friendly but if you are going to work for XYZ corporation, you will probably use uniform initialization.

Declaring and initializing user-defined types

We have already seen examples of how we declare and initialize some SFML-defined types. It is because the way that we can create/define these types (classes) is so flexible, that the way we declare and initialize them is also so varied. Here are a couple of reminders for declaring and initializing user-defined types from the previous chapter.

Create an object of type `VideoMode`, called `vm`, and initialize it with two `int` values, `1920` and `1080`.

```
// Create a video mode object
VideoMode vm(1920, 1080);
```

Create an object of the `Texture` type called `textureBackground`, but don't do any initialization.

```
// Create a texture to hold a graphic on the GPU
Texture textureBackground;
```

Note that it is possible (in fact, very likely) that even though we are not suggesting any specific values with which to initialize `textureBackground`, some setup of variables may take place internally. Whether or not an object needs or has the option of giving initialization values at this point is entirely dependent on how the class is coded and is almost infinitely flexible. This further suggests that when we get to write our own classes, there will be some complexity. Fortunately, it also means we will have significant power to design our types/classes to be just what we need to make our games. Add this huge C++ flexibility to the power of the SFML-designed classes and the potential for our games is almost limitless!

We will see a few more user-created types/classes provided by SFML in this chapter and loads more throughout the book. In *Chapter 6*, we will design

and code our own types/classes when implementing a Pong-style game.

Seeing how to manipulate the variables

At this point, we know exactly what variables are, the main types they can be, and how to declare and initialize them. We still haven't learned to achieve much with them, however. We need to manipulate our variables, add them, take away, multiply, divide, and especially, test them.

First, we will deal with how we can manipulate them and later we will look at how and why we test them.

With this in mind, let's learn about the C++ arithmetic and assignment operators.

C++ arithmetic and assignment operators

To manipulate variables, C++ has a range of **arithmetic operators** and **assignment operators**. Fortunately, most arithmetic and assignment operators are quite intuitive to use, and those that aren't are quite easy to explain. To get us started, let's look at a table of arithmetic operators followed by a table of assignment operators that we will regularly use throughout this book.

Arithmetic operator	Explanation

`+`	The addition operator can be used to add together the values of two variables or values.
`-`	The subtraction operator can be used to take away the value of one variable or value from another variable or value.
`*`	The multiplication operator can multiply the value of variables and values.
`/`	The division operator can divide the value of variables and values.
`%`	The **Modulo** operator divides a value or variable by another value or variable to find the remainder of the operation.

Table 2.2 Arithmetic operators

Now, for the assignment operators.

Assignment operators	**Explanation**
`=`	We have already seen this one. It is the assignment operator. We use it to initialize/set a variable's value.
`+=`	Add the value on the right-hand side to the variable on the left.
`-=`	Take away the value on the right-hand side from the variable on the left.
`*=`	Multiply by the value on the right-hand side by the variable on the left.

`/=`	Divide by the value on the right-hand side by the variable on the left.
`++`	**Increment** operator that adds one to a variable.
`--`	**Decrement** operator that takes away one from a variable.
`<=>`	The spaceship operator, represented by <=>, is a relatively new addition to the C++ language, introduced in C++20. It is used for three-way comparisons. We will explore this in a later project.

Table 2.3 Assignment operators

Technically, all the preceding operators, except for `=`, `--`, and `++`, are called **compound assignment operators** because they comprise more than one operator.

Now that we have seen a good range of arithmetic and assignment operators, we can see how to manipulate our variables by combining operators, variables, and values to form **expressions**.

Getting things done with expressions

Expressions are the combination of variables, operators, and values, just like expressions in English are the combination of words and punctuation. Using expressions, we can arrive at a result. Furthermore, as we will soon see, we can use an expression in a test. These tests can be used to decide what our code should do next.

Assignment

First, some simple expressions we might see in our game code.

```
// Player gets a new high score
hiScore = score;
```

Or,

```
// Set the score to 100
score = 100;
```

In the preceding code, we assign the value stored in score to the `hiScore` variable. From this point forward, `hiScore` will hold whatever was previously in `score`. We might do this at the end of a game when the player beats the previously held highest score. To be clear, we might go on to reset the score to zero and then use it to keep the score of the next game but `hiScore` will still hold the value previously stored in score when `hiScore = score` was executed. Of course, if we executed this line of code at the end of every game, we would run the risk of assigning a value to `hiScore` that wasn't a new highest score. This conundrum brings us back to the need for testing and comparing values. Let's keep going and we will get to the solution soon.

Next, look at the addition operator, which is used in conjunction with the assignment operator:

```
// Add to the score when an alien is shot
score = aliensShot + wavesCleared;
```

Or:

```
// Add 100 to whatever the score currently is
score = score + 100;
```

Notice that it is perfectly acceptable to use the same variable on both sides of an operator. In the preceding code, the first line assigns to `score` the result of adding the values in `aliensShot` and `wavesCleared` together. The second line of code assigns the value of whatever `score` currently holds plus one hundred back into `score`. Perhaps another variation of this example would be useful:

```
score = score + pointsPerAlien;
```

In this example, the value assigned to `pointsPerAlien` is added to the existing value in `score`. This technique of using variables on both sides of an operator is very common. Look at the code again and be sure you understand what is happening.

Next, let's look at the subtraction operator in conjunction with the assignment operator. The code that follows subtracts the value on the right side of the subtraction operator from the value on the left. It is usually used along with the assignment operator, perhaps like this:

```
// Uh oh lost a life
lives = lives - 1;
```

Or:

```
// How many aliens left at the end of game
aliensRemaining = aliensTotal - aliensDestroyed;
```

This is how we might use the division operator. This next code divides the number on the left by the number on the right. Again, it is usually used with the assignment operator, like this:

```
// Make the remaining hit points lower based on swordLevel
hitPoints = hitPoints / swordLevel;
```

Or:

```
// Give something, but not everything, back for recycling a bloc
recycledValueOfBlock = originalValue / 1.1f;
```

In the previous example, the `recycledValueOfBlock` variable will need to be of the `float` type to accurately store the answer to a calculation like that. Hopefully, this syntax is starting to seem obvious. If it seems like I am teaching you how to do child-level arithmetic, then that means you have probably got the gist. One more assignment operator example and we will move on.

Perhaps unsurprisingly, we could use the multiplication operator like this:

```
// answer is equal to 100, of course
answer = 10 * 10;
```

Or:

```
// biggerAnswer = 1000, of course
biggerAnswer = 10 * 10 * 10;
```

By now, the code probably doesn't need explaining. In the preceding examples, we are multiplying two and then, three instances of the number ten

together and assigning the results to `answer` and then `biggerAnswer` respectively.

Increment and decrement

Now, let's look at the increment operator in action. This is a neat way to add 1 to the value of one of our game's variables. Stay tuned for a C++ fun fact regarding the increment operator.

This code we have already seen and I do not need to explain it again, but just take another look.

```
// Add one to myVariable
myVariable = myVariable + 1;
```

Sometimes, it is not necessary to repeat the variable you want to use on both sides of an operator. It is possible to make your code clearer and save a tiny bit of typing time too.

This next code gives the same result as the previous code:

```
// Much neater, clearer and quicker
myVariable ++;
```

The increment operator is the exact same as the ++ in C++.

As a fun fact, have you ever wondered how C++ got its name? C++ is an extension of the C language. Its inventor, **Bjarne Stroustrup**, originally called it "C with classes" but the name evolved. If you are interested, read the C++ story: http://www.cplusplus.com/info/history/.

The decrement operator, `--`, is, you guessed it, a quick way to subtract 1 from something.

```
playerHealth = playerHealth -1;
```

This next code is quicker, clearer, and does the same thing as the preceding code:

```
playerHealth --;
```

Let's look at a few more operators in action and then we can get back to building the Timber!!! game. Try and work out what is happening in all of the lines of code that follow:

```
int someVariable = 10;
// Multiply the variable by 10 and put the answer
// back in the variable
someVariable *= 10;
// someVariable now equals 100
// Divide someVariable by 5 put the answer back
// into the variable
someVariable /= 5;
// someVariable now equals 20
// Add 3 to someVariable and put the answer back
// into the variable
someVariable += 3;
// someVariable now equals 23
// Take 25 from someVariable and put the answer back
// into the variable
someVariable -= 25;
// someVariable now equals -2
```

In the preceding code, we take the use of incrementing and decrementing to a new level using some compound operators that combine the assignment

operator with the increment and decrement operators. We are no longer just adding or subtracting one. When we use the `*=`, `/=`, `+=`, or `-=` operators, we are multiplying, dividing, adding, or subtracting the value currently held in the variable with the number preceding the operator.

So, in the multiplication example, `someVariable` holds the `10` value, and the code, `someVariable *= 10`, will multiply the value by `10` and put the answer, `100`, back into `someVariable`. This syntax is short, fast, and clear. Nice.

If any of these examples need further clarification, we will be reusing almost all we have just learned to enhance our game and get the graphics moving. It's time to add some more sprites to our game.

Adding clouds, a buzzing bee, and a tree

First, we will add a tree. This is going to be easy. The reason for this is because the tree doesn't move. We will use the same procedure that we used in the previous chapter when we drew the background. In this next section, we will prepare our static tree sprite and our moving bee and cloud sprites. We can then focus separately on moving and drawing the bee and the clouds because they will need a bit more C++ knowledge to do so.

Preparing the tree

Add the following highlighted code. Notice the un-highlighted code, which is the code we have already written. This should help you identify that the new code should be typed immediately after we set the position of the background but before the start of the main game loop. We will recap what is going on in the new code after you have added it.

```
int main()
{
// Create a video mode object
VideoMode vm(1920, 1080);
// Create and open a window for the game
RenderWindow window(vm, "Timber!!!", Style::Fullscreen);
// Create a texture to hold a graphic on the GPU
Texture textureBackground;
// Load a graphic into the texture
textureBackground.loadFromFile("graphics/background.png");
// Create a sprite
Sprite spriteBackground;
// Attach the texture to the sprite
spriteBackground.setTexture(textureBackground);
// Set the spriteBackground to cover the screen
spriteBackground.setPosition(0, 0);

// Make a tree sprite
Texture textureTree;
textureTree.loadFromFile("graphics/tree.png");
Sprite spriteTree;
spriteTree.setTexture(textureTree);
spriteTree.setPosition(810, 0);
while (window.isOpen())
{
```

The five lines of code (excluding the comment) that we just added, do the following:

- First, we create an object of the `Texture` type called `textureTree`.
- Next, we load a graphic into the texture from the `tree.png` graphics file.
- Then, we declare an object of type `Sprite` called `spriteTree`.
- Following on, we associate `textureTree` with `spriteTree`. Whenever we draw `spriteTree`, it will show the `textureTree` texture, which is a neat tree graphic.
- Finally, we set the position of the tree using the coordinates `810` on the x-axis and `0` on the y-axis.

A point to note is that the coordinates, 810 and zero, used to position the tree are values that I have tested and work nicely in our chosen overall resolution. I assigned the values in the manner that I did to quickly move on to the next subject. In a "real" C++ program, you would probably assign values to variables as this would make their uses clearer. Furthermore, if the values do not change, and they don't, you would probably use a constant variable as we discussed previously. You could declare variables like this outside the game loop:

```
const float TREE_HORIZONTAL_POSITION = 810;
const float TREE_VERTICAL_POSITION = 0;
```

Then, in the line of code where we draw the tree sprite, you would use the following:

```
spriteTree.setPosition(TREE_HORIZONTAL_POSITION, TREE_VERTICAL_P
```

In this example, the declaration is right before the usage, and I think that the `setPosition` function makes it clear enough what the values refer to.

I leave it as an exercise for the reader should they wish to modify the code if they think the effort of the two new constant variables will make things clearer. When we write code with unexplained values as we have, it is sometimes critically referred to as using **magic numbers** because they do something that is sometimes less clear than using a variable with a meaningful name. The point of this conversation is that the bigger and more complex your code is, the stricter you should be with your standards, especially if you are collaborating or being paid to be strict. I will occasionally use magic numbers for brevity but hopefully, the context will always be clear.

Let’s move on to the much more interesting bee.

Preparing the bee

The difference between this next code and the tree code is small but important. As the bee needs to move, we also declare two bee-related variables. Add the highlighted code in the place shown and see whether you can work out how we might use the `beeActive` and `beeSpeed` `variables`.

```
// Make a tree sprite
Texture textureTree;
textureTree.loadFromFile("graphics/tree.png");
Sprite spriteTree;
spriteTree.setTexture(textureTree);
spriteTree.setPosition(810, 0);
// Prepare the bee
Texture textureBee;
textureBee.loadFromFile("graphics/bee.png");
Sprite spriteBee;
spriteBee.setTexture(textureBee);
spriteBee.setPosition(0, 800);
// Is the bee currently moving?
bool beeActive = false;
// How fast can the bee fly
float beeSpeed = 0.0f;
while (window.isOpen())
{
```

In the preceding new code, we create a bee in the same way we created a background and a tree. We use a `Texture` and a `Sprite` and associate the two.

Note also in the previous bee code some new code we haven’t seen before in our project although we have just talked about it when discussing variables. There is a `bool` variable for determining whether the bee is active or not.

Remember that a `bool` variable can be either true or false. We initialize `beeActive` to `false`, for now.

Next, we declare a new `float` variable called `beeSpeed`. This will hold the speed that our bee will fly across the screen at in pixels per second.

Soon, we will see how we use these two new variables to move the bee. Before we do, let's set up some clouds in an almost identical manner.

Preparing the clouds

Add the highlighted code shown next. Study the new code and try and work out what it will do – and then I'll explain it.

```
// Prepare the bee
Texture textureBee;
textureBee.loadFromFile("graphics/bee.png");
Sprite spriteBee;
spriteBee.setTexture(textureBee);
spriteBee.setPosition(0, 800);
// Is the bee currently moving?
bool beeActive = false;
// How fast can the bee fly
float beeSpeed = 0.0f;
// make 3 cloud sprites from 1 texture
Texture textureCloud;
// Load 1 new texture
textureCloud.loadFromFile("graphics/cloud.png");
// 3 New sprites with the same texture
Sprite spriteCloud1;
Sprite spriteCloud2;
Sprite spriteCloud3;
spriteCloud1.setTexture(textureCloud);
spriteCloud2.setTexture(textureCloud);
spriteCloud3.setTexture(textureCloud);
// Position the clouds on the left of the screen
// at different heights
spriteCloud1.setPosition(0, 0);
```

```
spriteCloud2.setPosition(0, 250);
spriteCloud3.setPosition(0, 500);
// Are the clouds currently on screen?
bool cloud1Active = false;
bool cloud2Active = false;
bool cloud3Active = false;
// How fast is each cloud?
float cloud1Speed = 0.0f;
float cloud2Speed = 0.0f;
float cloud3Speed = 0.0f;
while (window.isOpen())
{
```

The only thing about the code we have just added that might seem a little odd is that we have only one object of the `Texture` type. It is completely normal for multiple `Sprite` objects to share a texture. Once a `Texture` is stored in GPU memory, it can be associated with a `Sprite` object very quickly. It is only the initial loading of the graphic in the `loadFromFile` code that is a relatively slow operation. Of course, if we wanted three differently shaped clouds, then we would need three textures.

Apart from the minor texture-sharing anomaly, the code we have just added is nothing new compared to the bee. The only difference is that there are three cloud sprites, three `bool` variables to determine whether each cloud is active, and three `float` variables to hold the speed for each cloud.

Drawing the tree, the bee, and the clouds

Finally, we can draw them all on the screen by adding this highlighted code in the drawing section.

```
/*
****************************************
Draw the scene
****************************************
*/
// Clear everything from the last run frame
window.clear();
// Draw our game scene here
window.draw(spriteBackground);
// Draw the clouds
window.draw(spriteCloud1);
window.draw(spriteCloud2);
window.draw(spriteCloud3);
// Draw the tree
window.draw(spriteTree);
// Draw the insect
window.draw(spriteBee);
// Show everything we just drew
window.display();
```

Drawing the three clouds, the bee, and the tree is done in the same way that the background was drawn. Notice, however, the order in which we draw the different objects to the screen. We must draw all the graphics after the background, or they will be obscured by the background, and we must draw the clouds before the tree, or they will look a bit odd drifting in front of the tree. The bee would look OK either in front or behind the tree. I opted to draw the bee in front of the tree, so it can try and distract our lumberjack, a bit like a real bee might.

Run Timber!!! and gaze in awe at the tree, the bee, and the three clouds that don't do anything! They look like they are lining up for a race; a race where the bee must go backward.

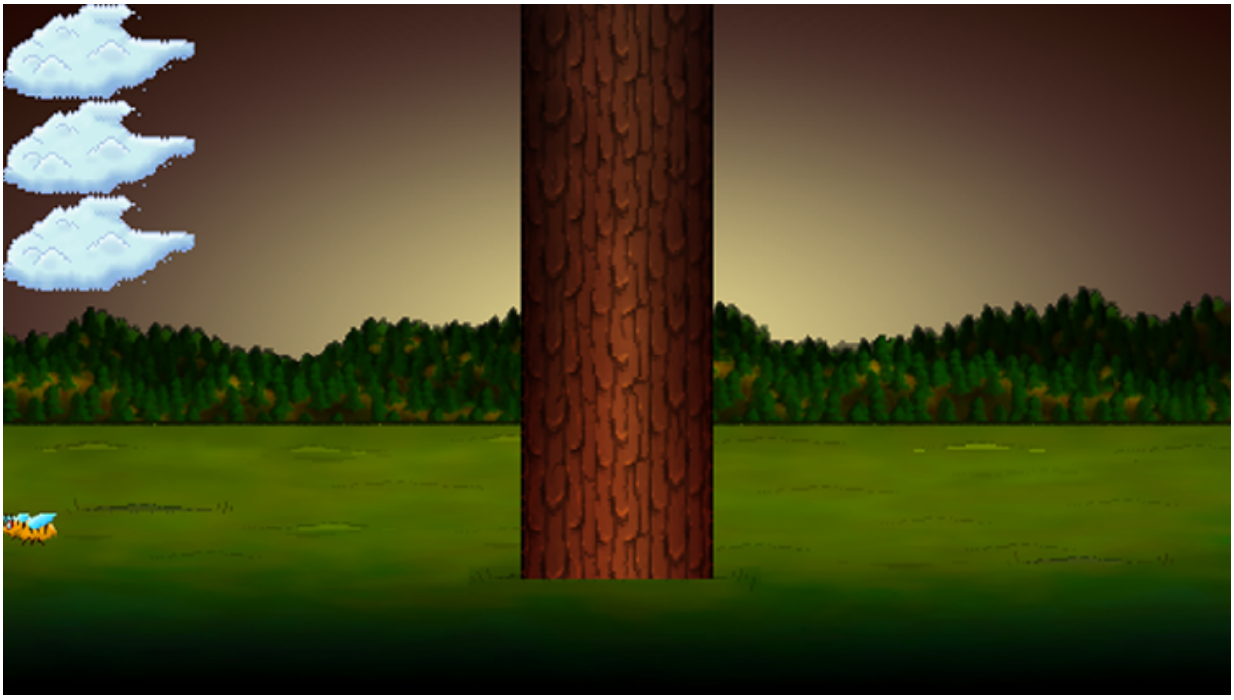


Figure 2.1: Drawing the tree, bee and clouds

Using what we know about C++ operators, we could try and move the graphics we have just added, but there are a couple of problems. Firstly, real clouds and bees move in a non-uniform manner. They don't have a set speed or location. Although their locations and speed are determined by factors such as the wind or how much of a hurry the bee might be in. To the casual observer, the path they take and their speed appear *random*. Let's explore randomness further.

Random numbers

Random numbers are useful for lots of reasons in games, for example, determining what card the player is dealt or how much damage within a certain range is subtracted from an enemy's health. As hinted at, we will use random numbers to determine the starting location and speed of the bee and the clouds.

Generating random numbers in C++

To generate random numbers, we will need to use some more C++ functions. Don't add any code to the game yet. Let's just look at the syntax and the steps required with some hypothetical code.

Computers can't genuinely pick random numbers. They can only use **algorithms** to pick a number that *appears* to be random. So that this algorithm doesn't constantly return the same value, we must **seed** the **random number generator**. The seed can be any integer number, although it must be a different seed each time you require a unique random number. Look at this code, which seeds the random number generator.

```
// Seed the random number generator with the time
srand((int)time(0));
```

The preceding code gets the time from the PC using the `time` function like this: `time(0)`. The call to the `time` function is enclosed as the value to be sent to the `srand` function. The result of this is that the current time is used as the seed.

The previous code is made to look a little more complicated because of the slightly unusual-looking `(int)` syntax. What this does is convert/cast the value returned from `time` to an `int`. This is required by the `srand` function in this situation.

The term used to describe a conversion from one type to another is **cast**.

So, in summary, the previous line of code:

1. Gets the time using `time`.
2. Converts it to type `int`.
3. Sends this resulting value to `srand` which seeds the random number generator.

The time is, of course, always changing. This makes the `time` function a great way to seed the random number generator. However, think about what might happen if we seed the random number generator more than once and in such quick succession that `time` returns the same value. We will see and solve this problem when we animate our clouds.

At this stage, we can create the random number between a range and save it to a variable for later use using code like this:

```
// Get the random number & save it to a variable called number
int number = (rand() % 100);
```

Notice the odd-looking way we assign a value to `number`. By using the modulo operator (`%`) and the value of `100`, we are asking for the remainder after dividing the number returned from `rand` by 100. When you divide by 100, the highest number you can possibly have as a remainder is 99. The lowest number possible is 0. Therefore, the previous code will generate a number between 0 and 99 inclusive. This knowledge will be useful for generating a random speed and starting location for our bees and clouds.

We will do this soon, but we first need to learn how to make decisions in C++.

Making decisions with if and else

The C++ `if` and `else` keywords are what enable us to make decisions. We saw `if` in action in the previous chapter when we detected each frame whether the player had pressed the *Escape* key.

```
if (Keyboard::isKeyPressed(Keyboard::Escape))
{
window.close();
}
```

So far, we have seen how we can use arithmetic and assignment operators to create expressions. Now, we can see some new operators.

Logical operators

Logical operators are going to help us make decisions, by building expressions that can be tested for a value of either `true` or `false`. At first, this might seem like quite a narrow choice and insufficient for the kind of choices that might be needed in an advanced PC game. Once we dig a little deeper, we will see that you can make all the required decisions we will need, with just a few of the logical operators.

Here is a table of the most useful logical operators. Look at them and the associated examples, and then we will see how to put them to use.

Logical operator	**Name and example**
`==`	The **comparison operator** tests for equality and is either `true` or `false`. An expression like `(10 == 9)`, for example, is `false`. 10 is obviously not equal to 9.

`!`	This is the logical **NOT operator**. The expression `(! (2 + 2 == 5))`. This is true because 2 + 2 is NOT 5.
`!=`	This is another comparison operator but it is different from the `=` comparison operator. This tests whether something is not equal. For example, the expression `(10 != 9)` is `true`. 10 is not equal to 9.
`>`	Another comparison operator – there are a few more as well. This tests whether something is greater than something else. The expression `(10 > 9)` is `true`.
`<`	You guessed it. This tests for values less than something else. The expression `(10 < 9)` is `false`.
`>=`	This operator tests whether one value is greater than or equal to the other and if either is true, the result is `true`. For example, the expression `(10 >= 9)` is `true`. The expression `(10 >= 10)` is also `true`.
`<=`	Like the previous operator, this one tests for two conditions, but this time less than or equal to. The expression `(10 <= 9)` is `false`. The expression `(10 <= 10)` is `true`.
`&&`	This operator is known as **logical AND**. It tests two or more separate parts of an expression and **both parts** must be true for the result to be `true`. Logical AND is usually used in conjunction with the other operators to build more complex tests. The expression `((10 > 9) && (10 < 11))` is `true` because both parts are true, so the expression is true. The expression `((10 > 9) && (10 < 9))` is `false` because only one part of the expression is true and the other is `false`.

`||`	This operator is called **logical OR** and it is just like logical AND except that at least one of two or more parts of an expression need to be true for the expression to be `true`. Let's look at the last example we used but switch the `&&` for `||`. The expression `((10 > 9) || (10 < 9))` is now `true` because one part of the expression is true.

Table 2.4 Logical operators

Let's meet the C++ `if` and `else` keywords, which will enable us to put all these logical operators to good use.

C++ if and else

Let's make the previous examples less abstract. Meet the C++ `if` keyword. We will use `if` and a few operators along with a small story to demonstrate their use. Next is a made-up military situation that will hopefully be less abstract than the previous examples.

If they come over the bridge, shoot them!

The captain is dying and, knowing that his remaining subordinates are not very experienced, he decides to write a C++ program to convey his last orders for after he has died. The troops must hold one side of a bridge while awaiting reinforcements.

The first command the captain wants to make sure his troops understand is this:

"If they come over the bridge, shoot them!"

So, how do we simulate this situation in C++? We need a `bool` variable, `isComingOverBridge`. The next bit of code assumes that the `isComingOverBridge` variable has been declared and initialized to either `true` or `false`.

We can then use `if` like this:

```
if(isComingOverBridge)
{
// Shoot them
}
```

If the `isComingOverBridge` variable is equal to `true`, the code inside the opening and closing curly braces `{...}` will run. If not, the program continues after the `if` block and without running the code within it.

Else do this instead

The captain also wants to tell his troops to stay put if the enemy is not coming over the bridge.

Now, we can introduce another C++ keyword, `else`. When we want to explicitly do something when the `if` does not evaluate to `true`, we can use `else`.

For example, to tell the troops to stay put if the enemy is not coming over the bridge, we could write this code:

```
if(isComingOverBridge)
{
// Shoot them
}
else
{
```

```
// Hold position
}
```

The captain then realized that the problem wasn't as simple as he first thought. What if the enemy comes come over the bridge, but has too many troops? His squad would be overrun and slaughtered. So, he came up with this code (we'll use some variables as well this time):

```
bool isComingOverBridge;
int enemyTroops;
int friendlyTroops;
// Initialize the previous variables, one way or another
// Now the if
if(isComingOverBridge && friendlyTroops > enemyTroops)
{
// shoot them
}
else if(isComingOverBridge && friendlyTroops < enemyTroops)
{
// blow the bridge
}
else
{
// Hold position
}
```

The preceding code has three possible paths of execution. First, if the enemy is coming over the bridge and the friendly troops are greater in number:

```
if(isComingOverBridge && friendlyTroops > enemyTroops)
```

Second, if the enemy troops are coming over the bridge but outnumber the friendly troops:

```
else if(isComingOveBridge && friendlyTroops < enemyTroops)
```

Then, the third and final possible outcome, which will execute if neither of the others is `true` is captured by the final `else`, without an `if` condition.

Reader challenge

Can you spot a flaw with the preceding code? One that might leave a bunch of inexperienced troops in complete disarray? The possibility of the enemy troops and friendly troops being exactly equal in number has not been handled explicitly and would therefore be handled by the final `else`. The final `else` is meant for when there are no enemy troops. I guess any self-respecting captain would expect his troops to fight in this situation. He could change the first `if` statement to accommodate this possibility.

```
if(isComingOverBridge && friendlyTroops >=  enemyTroops)
```

Finally, the captain's last concern was that if the enemy came over the bridge waving the white flag of surrender and were promptly slaughtered, then his men would end up as war criminals. The C++ code needed was obvious. Using the `wavingWhiteFlag` Boolean variable, he wrote this test:

```
if (wavingWhiteFlag)
{
// Take prisoners
}
```

But where to put this code was less clear. In the end, the captain opted for the following nested solution and changing the test for `wavingWhiteFlag` to logical NOT, like this:

```
if (!wavingWhiteFlag)
{
```

```
// not surrendering so check everything else
if(isComingOverTheBridge && friendlyTroops >= enemyTroops)
{
// shoot them
}
else if(isComingOverTheBridge && friendlyTroops < enemyTroops)
{
// blow the bridge
}
}
else
{
// this is the else for our first if
// Take prisoners
{
// Holding position
```

This demonstrates that we can nest `if` and `else` statements inside of one another to create quite deep and detailed decisions.

We could go on making more and more complicated decisions with `if` and `else` but what we have seen is more than enough of an introduction. It is probably worth pointing out that very often there is more than one way to arrive at a solution to a problem. The *right* way will usually be the way that solves the problem in the clearest and simplest manner.

We are getting closer to having all the C++ knowledge we need to be able to animate our clouds and bee. There is one final animation issue to discuss and then we can get back to the game.

Timing

Before we can move the bee and the clouds, we need to consider timing. As we already know, the main game loop executes repeatedly until the player presses the *Escape* key.

We have also learned that C++ and SFML are exceptionally fast. In fact, my modest laptop executes a simple game loop (like the current one) at around five thousand times per second. With this in mind, let's discuss the problem of making the rate at which each frame of animation is shown consistent and predetermined.

The frame rate problem

Let's consider the speed of the bee. For discussion, we could pretend that we are going to move it at 200 pixels per second. On a screen that is 1920 pixels wide, it would take, approximately, 10 seconds to cross the entire width, because 10 x 200 is 2000 (near enough to 1920).

Furthermore, we know that we can position any of our sprites with `setPosition(...,...)`. We just need to put the x and y coordinates in the parentheses.

In addition to setting the position of a sprite, we can also get the current position of a sprite. To get the horizontal x coordinate of the bee, for example, we would use this code:

```
float currentPosition = spriteBee.getPosition().x;
```

The current x (horizontal) coordinate of the bee is now stored in `currentPosition`. To move the bee to the right, we could then add the appropriate fraction of 200 (our intended speed) divided by 5000 (the approximate frames per second on my laptop) to `currentPosition`, like this:

```
currentPosition += 200/5000;
```

Now, we could use `setPosition` to move our bee. It would smoothly move from left to right by 200 divided by 5000 pixels in each frame. But there are two problems with this approach.

The frame rate is the number of times per second that our game loop is processed. That is, the number of times that we handle the player's input, update the game objects, and draw them to the screen. We will expand on and discuss matters of frame rate now and throughout the rest of the book.

The frame rate on my laptop might not always be constant. The bee might look like it is intermittently "boosting" its way across the screen as each frame executes at an inconsistent rate.

Of course, we want a wider audience for our game than just my laptop! Every PC's frame rate will vary, at least slightly. If you have an old PC, the bee will appear to be weighed down with lead, and if you have the latest gaming rig, it will probably be something of a blurry turbo bee.

Fortunately, this problem is the same for every game and SFML has provided a neat C++ solution. The easiest way to understand the solution is to implement it.

The SFML frame rate solution

We will now measure and use the frame rate to control our game. To get started implementing this, add this code just before the main game loop:

```
// How fast is each cloud?
float cloud1Speed = 0;
float cloud2Speed = 0;
float cloud3Speed = 0;
// Variables to control time itself
Clock clock;
```

```
while (window.isOpen())
{
```

In the previous code, we declare an object of the `Clock` type and we name it `clock`. The class name starts with a capital letter and the object name (that we will use) starts with a lowercase letter. The object name is arbitrary, but `clock` seems like an appropriate name for, well, a `clock`. We will add some more time-related variables here soon as well.

Now, in the update section of our game code, add this highlighted code:

```
/*
****************************************
Update the scene
****************************************
*/
// Measure time
Time dt = clock.restart();
/*
****************************************
Draw the scene
****************************************
*/
```

The `clock.restart()` function, as you might expect, restarts the `clock`. We want to restart the `clock` every frame so that we can time how long each frame takes. In addition, however, it returns the amount of time that has elapsed since the last time we restarted the `clock`.

As a result of this, in the previous code, we are declaring an object of the `Time` type called `dt` and using it to store the value returned by the `clock.restart()` function.

Now, we have a `Time` object called `dt` that holds the amount of time that elapsed since the last time we updated the scene and restarted the `clock`.

Maybe you can see where this is going.

Let's add some more code to the game and then we will see what we can do with `dt`.

`dt` stands for **delta time**, which is the time between two updates.

What we will do with this clock is update our game engine functionality to take time into account. Now, our game loop could be visualized like this next image:

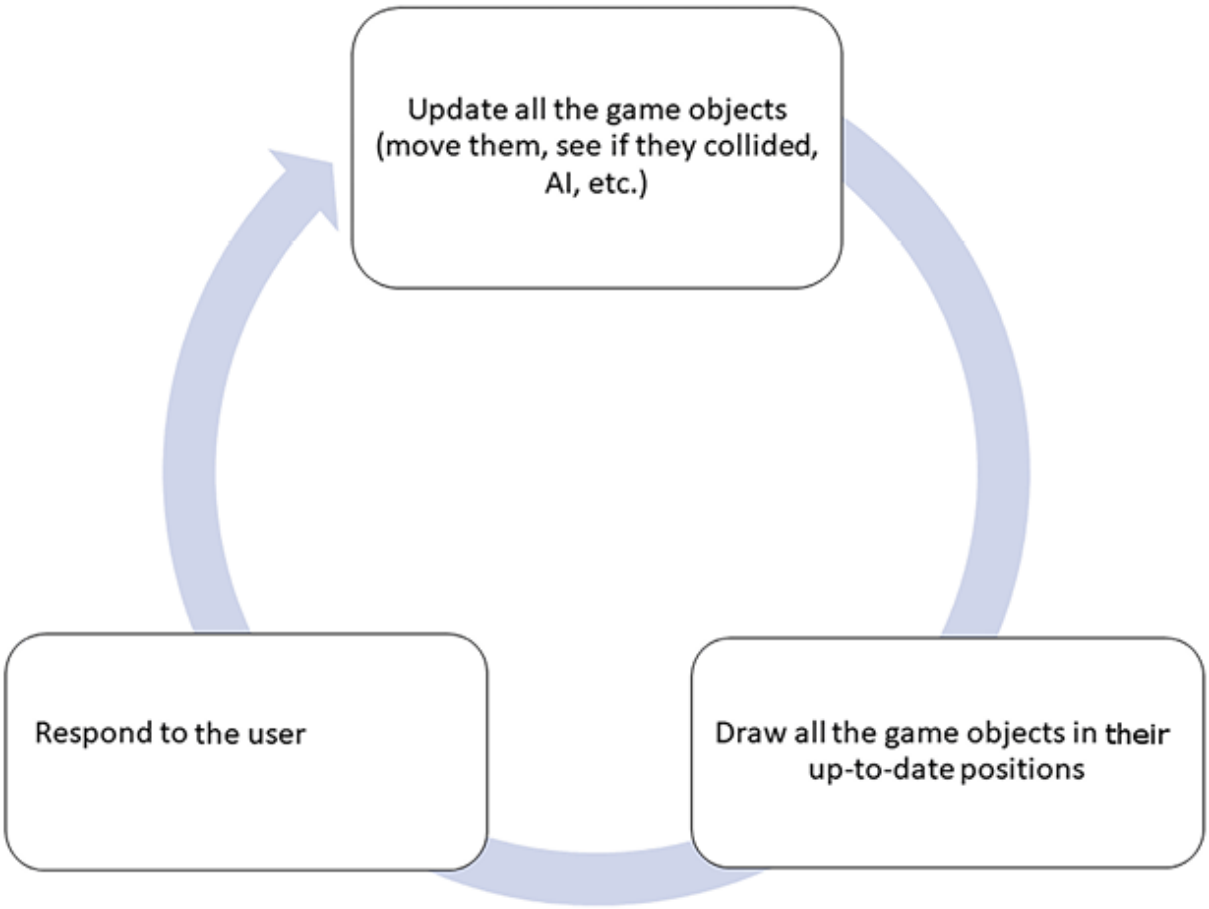

Update all the game objects (move them, see if they collided, AI, etc.)
Draw all the game objects in their up-to-date positions
Respond to the user

Figure 2.2: Basic game loop

With the introduction of the SFML `Clock` class, our game loop can be better represented with this next image:

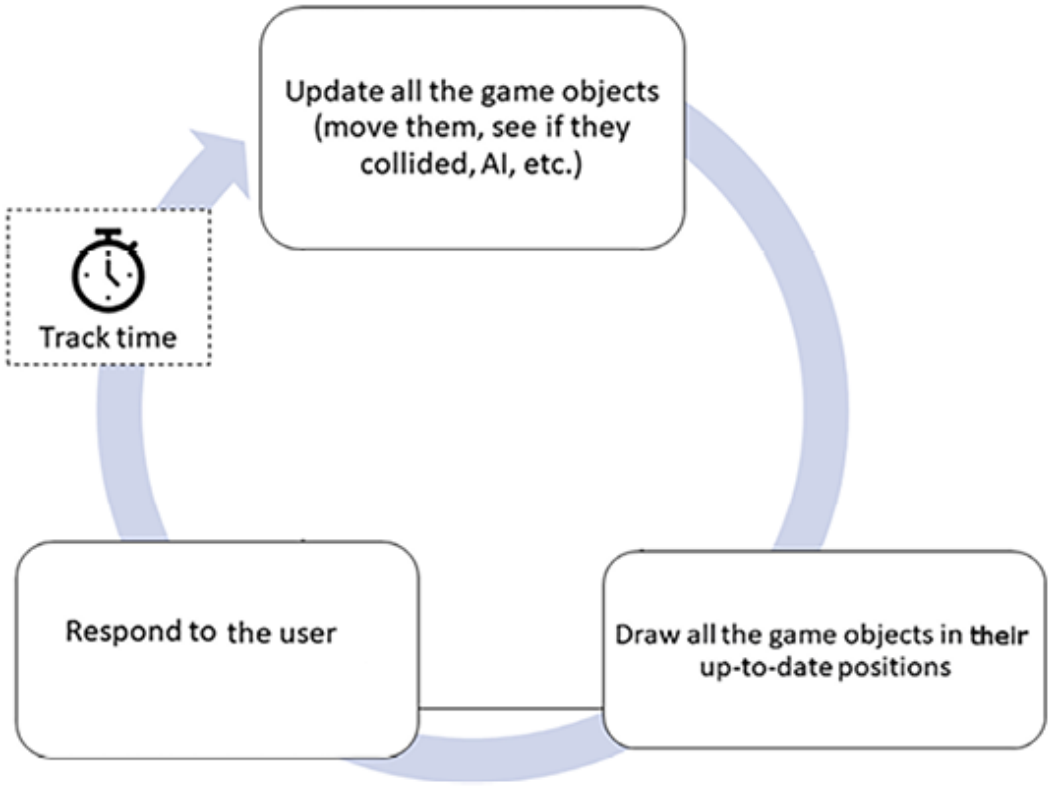

Update all the game objects (move them, see if they collided, AI, etc.)
Track time
Respond to the user
Draw all the game objects in their up-to-date positions

Figure 2.3: Basic game loop with timing

Let's add the key part of our timing code to see how the math works. Now, we can solve the problem of an inconsistent frame rate by updating the bees and the clouds relative to the amount of time that each frame takes to execute. If the frame is fast, we move the bee less than if the frame is slower.

Moving the clouds and the bee

Let's use the elapsed time since the last frame to breathe life into the bee and the clouds. This will solve the problem of needing to achieve a consistent frame rate across different PCs.

Giving life to the bee

The first thing we want to do is set up the bee at a certain height and a certain speed. We only want to do this when the bee is inactive. So, we wrap the next code in an `if` block. Examine and add the highlighted code, and then we will discuss it.

```
/*
****************************************
Update the scene
****************************************
*/
// Measure time
Time dt = clock.restart();
// Setup the bee
if (!beeActive)
{
// How fast is the bee
srand((int)time(0));
beeSpeed = (rand() % 200) + 200;
// How high is the bee
srand((int)time(0) * 10);
float height = (rand() % 500) + 500;
spriteBee.setPosition(2000, height);
beeActive = true;
}
/*
****************************************
Draw the scene
****************************************
*/
```

Now, if the bee is not active, just like it won't be when the game first starts, `if(!beeActive)` will be `true` and the preceding code will do the following things, in this order:

1. Seed the random number generator.

2. Get a random number between 200 and 399 and assign the result to `beeSpeed`.
3. Seed the random number generator again.
4. Get a random number between 500 and 999 and assign the result to a new `float` variable called `height`.
5. Set the position of the bee to `2000` on the x-axis (just off-screen to the right) and to whatever, `height`, equals on the y-axis.
6. Set `beeActive` to true so this code doesn't execute again until we again change `beeActive` later in the code.

Note that the `height` variable is the first variable we have ever declared inside the game loop. Furthermore, because it was declared inside an `if` block, it is "invisible" outside of the `if` block. This is fine for our use because once we have set the height of the bee, we don't need it anymore. This phenomenon that affects variables is called **scope**. We will explore this more fully in *Chapter 4*, *Loops, Arrays, Switch, Enumerations, and Functions: Implementing Game Mechanics*.

If we run the game, nothing will happen to the bee yet, but now that the bee is active, we can write some code that runs when `beeActive` is `true`.

Add the following highlighted code, which, as you can see, executes whenever `beeActive` is `true`. This is because it follows with an `else` after the `if(!beeActive)` block.

```
// Set up the bee
if (!beeActive)
{
```

```
// How fast is the bee
srand((int)time(0) );
beeSpeed = (rand() % 200) + 200;
// How high is the bee
srand((int)time(0) * 10);
float height = (rand() % 1350) + 500;
spriteBee.setPosition(2000, height);
beeActive = true;
}
else
// Move the bee
{
spriteBee.setPosition(
spriteBee.getPosition().x -
(beeSpeed * dt.asSeconds()),
spriteBee.getPosition().y);
// Has the bee reached the left-hand edge of the screen?
if (spriteBee.getPosition().x < -100)
{
  // Set it up ready to be a whole new bee next frame
    beeActive = false;
 }
}
/*
****************************************
Draw the scene
****************************************
*/
```

In the `else` block, the following things happen.

The bee position is changed using the following criteria. The `setPosition` function uses the `getPosition` function to get the current horizontal coordinate of the bee. It then subtracts `beeSpeed * dt.asSeconds()` from that coordinate.

The `beeSpeed` variable value is many pixels per second and was randomly assigned in the previous `if` block. The value of `dt.asSeconds()` will be a

fraction of 1, which represents how long the previous frame of animation took.

Assume that the bee's current horizontal coordinate is **1000**. Now, suppose a basic PC loops at 5000 frames per second. This would mean that `dt.asSeconds` would be **0.0002**. Further, suppose that `beeSpeed` was set to the maximum **399** pixels per second. Then, the code that determines the value that `setPosition` uses for the horizontal coordinate can be explained like this:

```
1000 - 0.0002 x 399
```

Therefore, the new position on the horizontal axis for the bee would be **999.9202**. We can see that the bee is very, very smoothly drifting to the left, well under a pixel per frame. If the frame rate fluctuates, then the formula will produce a new value to suit. If we run the same code on a PC that only achieves 100 frames per second or a PC that achieves a million frames per second, the bee will move at the same speed.

The `setPosition` function uses `getPosition().y` to keep the bee in exactly the same vertical coordinate throughout this cycle of being active.

The final part of the code in the `else` block we just added is shown again so we can talk about it next:

```
// Has the bee reached the right hand edge of the screen?
if (spriteBee.getPosition().x < -100)
{
  // Set it up ready to be a whole new bee next frame
  beeActive = false;
}
```

This code tests, each frame (when `beeActive` is `true`), whether the bee has disappeared off the left-hand side of the screen. If the `getPosition` function returns less than -100, it will certainly be out of view of the player. When this occurs, `beeActive` is set to `false`, and, on the next frame, a "new" bee will be set to flying at a new random height and a new random speed.

Try running the game and watch our bee dutifully fly from right to left and then recycle itself back to the right again at a new height and speed. It's almost like a new bee every time.

Of course, a real bee would stick around for ages and pester you while you're trying to concentrate on chopping the tree. Also, a real bee would probably vary its height. Don't worry, we will be making more advanced game objects with each project. The point is that you should recycle/reuse your sprites and textures whenever possible for a more sustainable video game.

Now, we will get the clouds moving in a very similar way.

Blowing the clouds

The first thing we want to do is set up the first cloud at a certain height and a certain speed. We only want to do this when the cloud is inactive. Consequently, we will wrap the next code in another `if` block. Examine and add the highlighted code, just after the code we added for the bee, then we will discuss it. It has many similarities to the code we used on the bee.

```
else
// Move the bee
```

```
  {
    spriteBee.setPosition(
    spriteBee.getPosition().x -
    (beeSpeed * dt.asSeconds()),
    spriteBee.getPosition().y);
    // Has the bee reached the right hand edge of the screen?
    if (spriteBee.getPosition().x < -100)
    {
    // Set it up ready to be a whole new bee next frame
      beeActive = false;
    }
  }
  // Manage the clouds
  // Cloud 1
  if (!cloud1Active)
  {

    // How fast is the cloud
    srand((int)time(0) * 10);
    cloud1Speed = (rand() % 200);

    // How high is the cloud
    srand((int)time(0) * 10);
    float height = (rand() % 150);
    spriteCloud1.setPosition(-200, height);
    cloud1Active = true;
  }
  /*
  ****************************************
  Draw the scene
  ****************************************
  */
```

The only difference between the code we have just added and the bee-related code is that we work on a different sprite and use different ranges for our random numbers. Also, we use `* 10` to the result returned by `time(0)` so we are always guaranteed to get a different seed for each of the clouds. When we code the other cloud movement, you will see that we use `* 20` and `* 30` respectively.

Now, we can act when the cloud is active. We will do so in the `else` block. As with the `if` block, the code is identical to that of the bee-related code, except that all the code works on the cloud instead of the bee.

```
// Manage the clouds
if (!cloud1Active)
{
  // How fast is the cloud
  srand((int)time(0) * 10);
  cloud1Speed = (rand() % 200);

  // How high is the cloud
  srand((int)time(0) * 10);
  float height = (rand() % 150);
  spriteCloud1.setPosition(-200, height);
  cloud1Active = true;

}
else
 {
   spriteCloud1.setPosition(
   spriteCloud1.getPosition().x +
   (cloud1Speed * dt.asSeconds()),
   spriteCloud1.getPosition().y);

   // Has the cloud reached the right hand edge of the screen?
   if (spriteCloud1.getPosition().x > 1920)
   {
     // Set it up ready to be a whole new cloud next frame
     cloud1Active = false;
   }
 }
/*
****************************************
Draw the scene
****************************************
*/
```

Now that we know what to do, we can duplicate the same code for the second and third cloud. Add this highlighted code, which handles the second and third cloud, immediately after the code for the first cloud:

```
...
// Cloud 2
if (!cloud2Active)
{

  // How fast is the cloud
  srand((int)time(0) * 20);
  cloud2Speed = (rand() % 200);

  // How high is the cloud
  srand((int)time(0) * 20);
  float height = (rand() % 300) - 150;
  spriteCloud2.setPosition(-200, height);
  cloud2Active = true;


}
else
{

  spriteCloud2.setPosition(
  spriteCloud2.getPosition().x +
  (cloud2Speed * dt.asSeconds()),
  spriteCloud2.getPosition().y);

  // Has the cloud reached the right hand edge of the screen?
  if (spriteCloud2.getPosition().x > 1920)
  {
    // Set it up ready to be a whole new cloud next frame
    cloud2Active = false;
  }
}
if (!cloud3Active)
{

  // How fast is the cloud
  srand((int)time(0) * 30);
```

```
    cloud3Speed = (rand() % 200);

    // How high is the cloud
    srand((int)time(0) * 30);
    float height = (rand() % 450) - 150;
    spriteCloud3.setPosition(-200, height);
    cloud3Active = true;


}
else
{

    spriteCloud3.setPosition(
    spriteCloud3.getPosition().x +
    (cloud3Speed * dt.asSeconds()),
    spriteCloud3.getPosition().y);

    // Has the cloud reached the right hand edge of the screen?
    if (spriteCloud3.getPosition().x > 1920)
    {
      // Set it up ready to be a whole new cloud next frame
      cloud3Active = false;
    }
}
/*
****************************************
Draw the scene
****************************************
*/
```

Now, you can run the game and the clouds will randomly and continuously drift across the screen and the bee will buzz from right to left before re-spawning once more back on the right.

Figure 2.4: Blowing the clouds

Does all this cloud and bee handling seem a little bit repetitive? We will see how we can save lots of typing and make our code more readable. In C++, there are ways of handling multiple instances of the same type of variable or object. These are called **arrays** and we will learn about them in *Chapter 4*, *Loops, Arrays, Switch, Enumerations, and Functions: Implementing Game Mechanics*. Furthermore, we will also see how we can execute the same code but on different values without writing that code multiple times (as we have done here) using our own custom written functions. All this time-saving efficiency will be explored in *Chapter 4*. It was a deliberate choice to prioritize progress with the game features rather than introducing even more C++ before making more progress. By the end of this book, you will

> know how to make this game much better than we can at the moment.

Before you move on, I encourage you to play with the code from this chapter. How about swapping the texture files for your own images, changing the speed of the bee and the clouds, or making the bee go up and down in a kind of sine wave across the screen? Look at a few frequently asked questions related to the topics in this chapter.

Summary

In this chapter, we learned that a variable is a named storage location in memory, in which we can keep values of a specific type. Types include `int`, `float`, `double`, `bool`, `String`, and `char`.

We can declare and initialize all the variables we need to store the data for our game. Once we have our variables, we can manipulate them using the arithmetic and assignment operators as well as use them in tests with the logical operators. Used in conjunction with the `if` and `else` keywords, we can branch execution of our code depending upon the current situation in the game.

Using all this new knowledge, we animated some clouds and a bee. In the next chapter, we will use these skills some more to add a **heads-up display** (**HUD**) and add more input options for the player, as well as represent time visually using a time bar.

Frequently Asked Questions

Q) Why do we set the bee to inactive when it gets to -100? Why not just zero because zero is the left-hand side of the window?

A) The bee graphic is 60 pixels wide and its origin is at the top left pixel. As a result, when the bee is drawn with its origin at x equals zero, the entire bee graphic is still on screen for the player to see. By waiting until it is at -100, we can be sure it is out of the player's view.

Q) How do I know how fast my game loop is?

A) If you have a modern NVIDIA graphics card you might be able to already by configuring your GeForce Experience overlay to show the frame rate. To measure this explicitly using our own code, however, we will need to learn a few more things. We will add the ability to measure and display the current frame rate in *Chapter 5*, *Collisions, Sound, and End Conditions: Making the Game Playable*.

Q) What is the difference between the assignment operator, `=`, and the equality operator, `==`, in C++?

A) The assignment operator, `=`, is used to assign a value to a variable. For example, `int x = 5` assigns the value `5` to the variable `x`. The equality operator, `==`, is used to compare two values for equality. For example, `if (x == 5)` checks whether the value of `x` is equal to `5`.

Q) How do sprites and textures work together in C++ with SFML?

A) In SFML, a Texture represents an image loaded from a file, while a Sprite is a 2D image that can be drawn on the screen. The `setTexture` function associates a Texture with a Sprite, enabling the rendering of the image on the screen. You can manipulate the sprite's position, rotation, and scale, and SFML handles the rendering efficiently using the GPU.

Q) What is the purpose of seeding the random number generator when generating random numbers in C++?

A) Seeding the random number generator is essential to ensure that it produces different sequences of random numbers each time the program runs. Without seeding, the generator would produce the same sequence of numbers on each program run, making the results predictable rather than random. Typically, the current time is used as the seed for randomness. This is much the same as providing a seed to generate a unique map in a game such as Minecraft. Later, in the final project, we will use more advanced techniques to generate random numbers.

3

C++ Strings, SFML Time: Player Input and HUD

Almost every game ever made will need to have some text on the screen – the score, the text of a character's speech, and many other examples. Therefore, in this chapter, we will spend around half the time learning how to manipulate text and display it on the screen and the other half looking at timing and how a visual time-bar can inform the player of their remaining time and create a sense of urgency in the game.

We will cover the foallowing:

- Pausing and restarting the game
- C++ strings
- SFML Text and SFML Font
- Adding a score and a message
- Adding a time-bar

As we progress with this game over the next three chapters, the code will get longer and longer. So, now seems like a good time to think ahead and add a little bit more structure to our code. We will add this structure to give us the ability to pause and restart the game.

Pausing and restarting the game

We will add code so that when the game is first run, it will be in a **paused** state. The player will then be able to press the *Enter* key to start the game. Then, the game will run until either the player gets squashed or runs out of time.

At this point, the game will pause once more and wait for the player to press *Enter* to restart again.

Let's step through setting this up a bit at a time. First, declare a new `bool` variable called `paused`, outside the main game loop, and initialize it to `true`.

```
// Variables to control time itself
Clock clock;
// Track whether the game is running
bool paused = true;
while (window.isOpen())
{
/*
****************************************
Handle the players input
****************************************
*/
```

Now, whenever the game is run, we have a variable, `paused`, that will be `true`.

Next, we will add another `if` statement where the expression will check to see whether the *Enter* key is currently being pressed. If it is being pressed, it sets `paused` to `false`. Add the highlighted code just after our other keyboard handling code.

```
/*
****************************************
Handle the players input
****************************************
```

```
*/
if (Keyboard::isKeyPressed(Keyboard::Escape))
{
window.close();
}
// Start the game
if (Keyboard::isKeyPressed(Keyboard::Return))
{
paused = false;
}
/*
****************************************
Update the scene
****************************************
*/
```

Now we have a `bool` called `paused`, which starts as `true` but changes to `false` when the player presses the *Enter* key. At this point, we must make our game loop respond appropriately, based on whatever the current value of `paused` might be.

This is how we will proceed. We will wrap the entire update part of the code, including the code we wrote in the last chapter for moving the bee and clouds, in an `if` statement.

Notice in the next code that the `if` block will only execute when `paused` is not equal to `true`. Or, put another way, the game won't move/update when it is paused. We could also wrap the drawing code in a similar `if` statement and this would prevent the scene from being drawn to the screen. As we know, when most games are paused, the action is paused but the scene remains visible. This is exactly what we want.

Look carefully at the precise place to add the new `if` statement and its corresponding opening and closing curly braces, `{...}`. If they are put in the wrong place, things will not work as expected.

Add the highlighted code to wrap the update part of the code, paying close attention to the context shown next. I have added ellipses, `...`, on a few lines to represent the unshown code. Of course, the `...` is not real code and should not be added to the game. You can identify where to place the new code (highlighted) at the start and the end by the unhighlighted code surrounding it.

```
/*
******************************************
Update the scene
******************************************
*/
if (!paused)
{
// Measure time
...
...
...
// Has the cloud reached the right hand edge of the screen?
if (spriteCloud3.getPosition().x > 1920)
{
// Set it up ready to be a whole new cloud next frame
cloud3Active = false;
}
}
} // End if(!paused)
/*
******************************************
Draw the scene
******************************************
*/
```

Notice that when you place the closing curly brace of the new `if` block, Visual Studio neatly adjusts all the indenting to keep the code tidy. However, depending on your Visual Studio settings, this might not happen. If your code inside the `if` block does not indent to the right by one tab, you can

select all the code inside the `if` block by clicking and dragging just as you would in any text-based app and then tap the `Tab` key on the keyboard. Now your code should be neatly indented.

Now you can run the game, and everything will be static until you press the *Enter* key. It is now possible to go about adding features to our game; we just need to remember that when the player dies or runs out of time, we need to set the `paused` variable to `true`.

In the previous chapter, we had a first glimpse at C++ strings. We need to learn some more about them so we can implement the player's HUD.

C++ strings

In the previous chapter, we briefly mentioned strings, and we learned that a string can hold alphanumeric data: anything from a single character to a whole book. We didn't look at declaring, initializing, or manipulating strings. So, let's do that now.

Declaring strings

Declaring a string variable is simple. We state the type, followed by the name.

```
String levelName;
String playerName;
```

Once we have declared a String, we can assign a value to it.

Assigning a value to strings

To assign a value to a string, as with regular variables, we simply put the name, followed by the assignment operator, then the value.

```
levelName = "Dastardly Cave";
playerName = "John Carmack";
```

Note that the values need to be enclosed in quotation marks. As with regular variables, we can also declare and assign values in a single line.

```
String score = "Score = 0";
String message = "GAME OVER!!";
```

For completeness, I should mention you can also declare and initialize strings using uniform initialization, as we discussed in *Chapter 2*, as shown next:

```
// Using uniform initialization for a string
string playerName{"Rob Hubbard"};
```

Strings in C++ are essential for handling text-based data in game development. Whether it's displaying player names as just suggested, displaying messages, or keeping track of who achieved the highest score, understanding how to work with strings is useful. Let's explore this further, starting with string **concatenation**.

String Concatenation

In the next code sample, we use C++ `cout` to output text to the console window. You can try this out by copying and pasting the code into just inside the opening curly brace of the main function of our current project, or start a

new project if you want to keep it separate. If you create a new project, you do not need to add any of the SFML configuration that we did in *Chapter 1*. Just create a console app, choose a name, paste the code inside the main function, and add these two includes for the `string` and `cout` functionality: `#include <iostream>` and `#include <string>`. Here is the code; try it out or just look and then we will talk about it.

```
// Before the main function
#include <iostream>
#include <string>
// Inside the main function
std::string playerName = "Player1";
std::string message = "Welcome to the game, " + playerName + "!"
std::cout << message << std::endl;
```

In the preceding code, we demonstrate how to create and manipulate strings in C++. It initializes a variable called `playerName` and constructs a string called `message` that includes the player's name, which is then displayed on the screen using `std::cout`. Note that in the middle line we concatenate (join) strings using the + operator.

Note that as with `sf::` in SFML, you can omit all the `std::` instances by adding a line of code after your `include` directives like this:

```
using namespace std;
```

There is much more we can do with strings, so let's keep going.

Getting the string length

In the next code, we go further into the world of strings and use the `length` function. We are jumping ahead of ourselves a little as this demonstrates calling a function on an instance of a class, but as you can see, it is quite intuitive.

```
    string playerName = "Player1";
    int playerNameLength = playerName.length();
    cout << "Player name has " << playerNameLength << " characte
```

In the preceding code, I have omitted all the `std::` specifiers that were present in the previous example, so if you want to try this code out in Visual Studio, you will need to add the `using namespace std` syntax after the `include` directives.

In the preceding code, we declare and initialize both a `string` and an `int`. We then use the `length()` function to return the number of characters in the string and store that result in the `playerNameLength` variable, which is of type `int`. We then use `cout` to print the results to the console window.

It should be obvious that `<<` joins together the sections of output. `<<` is a **bitwise** operator, but you might like to know a bit more about it.

The `<<` operator is one of the bitwise operators. C++, however, allows you to write your own classes and **override** what a specific operator does, within the context of your class. The `iostream` class has done this to make the `<<` operator work the way it does. The complexity is hidden in the class. We can use its functionality without worrying about how it works.

We are nearly ready to add more features to our game. First, let's see how we can change our `String` variables another way.

Manipulating strings another way with StringStream

We can use the `#include <sstream>` directive to give us some extra power with our strings. The `sstream` class enables us to "add" some strings together. When we do so, it is another way to do **concatenation**.

```
String part1 = "Hello ";
String part2 = "World";
sstream ss;
ss << part1 << part2;
// ss now holds "Hello World"
```

In addition to this, using `sstream` objects, a `String` variable can even be concatenated with a variable of a different type. The next code starts to reveal how strings might be quite useful to us.

```
String scoreText = "Score = ";
int score = 0;
// Later in the code
score ++;
sstream ss;
ss << scoreText << score;
// ss now holds "Score = 1"
```

Now we know the basics of C++ strings and how we can use `sstream`, we can see how to use some SFML classes to display them on the screen.

SFML Text and SFML Font

Let's talk about the SFML `Text` and `Font` classes a bit with some hypothetical code, before we actually go ahead and add code to our game.

The first step in drawing text on the screen is to have a font. In the first chapter, we added a font file to the project folder. Now we can load the font, ready for use, into an SFML `Font` object.

The code to do so looks like this:

```
Font font;
font.loadFromFile("myfont.ttf");
```

In the previous code, we first declare a `Font` object and then load an actual font file into it. Note that `myfont.ttf` is a hypothetical font and we could use any font that is in the project folder.

Once we have loaded a font, we need an SFML `Text` object.

```
Text myText;
```

Now we can configure our `Text` object. This includes the size, the color, the position on screen, the `String` that holds the message, and, of course, associating it with our `font` object.

```
// Assign the actual message
myText.setString("Press Enter to start!");
// assign a size
myText.setCharacterSize(75);
// Choose a color
myText.setFillColor(Color::White);
// Set the font to our Text object
myText.setFont(font);
```

It is worth interjecting a little at this point. I could interject after introducing almost every single SFML class we have used so far. It is almost impossible to overstate just how much work SFML saves us with this fantastic library, and the `Font` and `Text` classes are two good examples of this.

What SFML is doing "under the hood" is providing very simplified abstractions for handling fonts and text rendering, making it significantly easier compared to dealing directly with **OpenGL** for these tasks.

The `Font` class in SFML represents a font that can be used for rendering text. It provides functions to load fonts from files, in-memory buffers, or system fonts. The `Text` class is responsible for rendering text using a given font. It encapsulates the string to be displayed, the font, and various text-related properties.

SFML abstracts away almost every complexity involved in rendering text with **OpenGL**. It handles texture creation, shader management, and other **OpenGL** details behind the scenes. Using SFML for text rendering massively simplifies the intricacies of using **OpenGL** directly. SFML allows us to focus more on the game rather than the low-level math of OpenGL.

SFML was created by Laurent Gomila. Development of SFML began around 2006 and it has undergone many updates and improvements over the years. Laurent's dedication, over approaching two decades, to maintaining SFML cannot be overstated. In my view, it's incredible. I just thought I would mention it so every time you effortlessly draw a sprite on the screen, you think of the tireless effort that has gone into this behind the scenes.

We now know more than enough to add some features to our game. Let's add an **HUD** to Timber!!!.

Adding a score and a message

Now we know enough about strings, SFML `Text`, and SFML `Font` to go about implementing the **HUD**. HUD stands for **heads-up display** and more formally refers to a cockpit instrumentation display that doesn't require the pilot to look down. However, video game user interfaces, especially in-game interfaces, are often referred to as a HUD because they serve the same purpose as a cockpit HUD.

The next thing we need to do is add another `#include` directive to the top of the code file. As we have learned, the `sstream` class adds some useful functionality for combining strings and other variable types together into a single String.

Add the line of highlighted code.

```
#include <sstream>
#include <SFML/Graphics.hpp>
using namespace sf;
int main()
{
```

Next, we will set up our SFML `Text` objects: one to hold a message that we will vary to suit the state of the game and one that will hold the score and need to be regularly updated.

The next code declares the `Text` and `Font` objects, loads the font, assigns the font to the `Text` objects, and then adds the `String` messages, color, and size. This should look familiar from our discussion in the previous section. In addition, we add a new `int` variable called `score` that we can manipulate to hold the player's score.

Remember that if you chose a different font to `KOMIKAP_.ttf`, back in *Chapter 1*, *Welcome to Beginning C++ Game Programming, Third Edition*, you will need to change that part of the code to match the `.ttf` file that you have in the `Visual Studio Stuff/Projects/Timber/fonts` folder.

Add the highlighted code and we will be ready to move on to updating the HUD.

```
// Track whether the game is running
bool paused = true;
// Draw some text
int score = 0;
Text messageText;
Text scoreText;
// We need to choose a font
Font font;
font.loadFromFile("fonts/KOMIKAP_.ttf");
// Set the font to our message
messageText.setFont(font);
scoreText.setFont(font);
// Assign the actual message
messageText.setString("Press Enter to start!");
scoreText.setString("Score = 0");
// Make it really big
messageText.setCharacterSize(75);
scoreText.setCharacterSize(100);
// Choose a color
messageText.setFillColor(Color::White);
scoreText.setFillColor(Color::White);
while (window.isOpen())
{
/*
****************************************
Handle the players input
```

```
        ****************************************
        */
```

The next code might look a little convoluted, even complex. It is, however, straightforward when you break it down a bit. Examine and add the new code, then we will go through it.

```
        // Choose a color
        messageText.setFillColor(Color::White);
        scoreText.setFillColor(Color::White);
        // Position the text
        FloatRect textRect = messageText.getLocalBounds();
        messageText.setOrigin(textRect.left +
        textRect.width / 2.0f,
        textRect.top +
        textRect.height / 2.0f);
        messageText.setPosition(1920 / 2.0f,1080 / 2.0f);
        scoreText.setPosition(20, 20);
        while (window.isOpen())
        {
        /*
        ****************************************
        Handle the players input
        ****************************************
        */
```

We have two objects of type `Text` that we will display on the screen. We want to position `scoreText` at the top left with a little bit of padding. This is not a challenge; we simply use `scoreText.setPosition(20, 20)` and that positions it at the top left with 20 pixels of horizontal and vertical padding.

Positioning `messageText`, however, is not so easy. We want to position it on the exact midpoint of the screen. Initially, this might not seem like a problem, but then we remember that the origin of everything we draw is the top left-hand corner. So, if we simply divide the screen width and height by

2 and use the results in `messageText.setPosition...`, then the top left of the text will be in the center of the screen and it will spread out untidily to the right.

What we need is a way to be able to set the center of `messageText` to the center of the screen. The rather complex-looking bit of code that you just added repositions the origin of `messageText` to the center of itself. Here is the code under current discussion again for convenience.

```
// Position the text
FloatRect textRect = messageText.getLocalBounds();
messageText.setOrigin(textRect.left +
textRect.width / 2.0f,
textRect.top +
textRect.height / 2.0f);
```

First in this code, we declare a new object of type `FloatRect`, called `textRect`. A `FloatRect` object, as the name suggests, holds a rectangle with floating-point coordinates.

The code then uses the `mesageText.getLocalBounds` function to initialize `textRect` with the coordinates of the rectangle that wraps `messageText`.

The next line of code, which is spread over four lines as it is quite long, uses the `messageText.setOrigin` function to change the origin (the point that is used to draw) to the center of `textRect`. Of course, `textRect` holds a rectangle that exactly matches the coordinates that wrap `messageText`. Then, this next line of code executes:

```
messageText.setPosition(1920 / 2.0f,1080 / 2.0f);
```

Now, `messageText` will be neatly positioned in the exact center of the screen. We will use this exact same code each time we change the text of `messageText`, because changing the message changes the size of `messageText`, so its origin will need recalculating.

Next, we declare an object of type `stringstream` called `ss`. Note that we use the full name including the namespace `std::stringstream`. We could avoid this syntax by adding `using namespace std` to the top of our code file. We don't, however, because we use it infrequently. Take a look at the code and add it to the game, then we can go through it in more detail. As we only want this code to execute when the game is not paused, be sure to add it with the other code, inside the `if(!paused)` block, as shown.

```
else
{
spriteCloud3.setPosition(
spriteCloud3.getPosition().x +
(cloud3Speed * dt.asSeconds()),
spriteCloud3.getPosition().y);
// Has the cloud reached the right hand edge of the screen?
if (spriteCloud3.getPosition().x > 1920)
{
// Set it up ready to be a whole new cloud next frame
cloud3Active = false;
}
}
// Update the score text
std::stringstream ss;
ss << "Score = " << score;
scoreText.setString(ss.str());
}// End if(!paused)
/*
****************************************
Draw the scene
****************************************
*/
```

We use `ss` and the special functionality provided by the `<<` operator, which concatenates variables into a `stringstream`. So, the code `ss << "Score = " << score` has the effect of creating a `String` with `"Score = "`, and whatever the value of `score` is is concatenated together. For example, when the game first starts, `score` is equal to 0, so `ss` will hold the value `"Score = 0"`. If `score` ever changes, `ss` will adapt at each frame.

The next line of code simply displays/sets the `String` contained in `ss` to `scoreText`.

```
scoreText.setString(ss.str());
```

It is now ready to be drawn to the screen.

This next code draws both `Text` objects (`scoreText` and `messageText`), but notice that the code that draws `messageText` is wrapped in an `if` statement. This `if` statement causes `messageText` to only be drawn when the game is paused.

Add the highlighted code shown next.

```
// Now draw the insect
window.draw(spriteBee);
// Draw the score
window.draw(scoreText);
if (paused)
{
// Draw our message
window.draw(messageText);
}
// Show everything we just drew
window.display();
```

We can now run the game and see our HUD drawn on the screen. You will see the **SCORE = 0** and **PRESS ENTER TO START!** messages. The latter will disappear when you press *Enter*.

Figure 3.1: HUD in action

If you want to see the score updating, add a temporary line of code, `score ++;`, anywhere in the `while(window.isOpen)` loop. If you add this temporary line, you will see the score go up fast – very fast!

Figure 3.2: Score

If you added the temporary code `score ++;`, be sure to delete it before continuing.

Adding a time-bar

As time is a crucial mechanic in the game, it is necessary to keep the player aware of it. They need to know if their allotted six seconds are about to run out. It will give them a sense of urgency as the end of the game draws near and a sense of accomplishment if they perform well enough to maintain or increase their remaining time.

A drawing of the number of seconds remaining on the screen is not easy to read (when concentrating on the branches) or a particularly interesting way to achieve the objective.

What we need is a time-bar. Our time-bar will be a simple red rectangle, prominently displayed on the screen. It will start off nice and wide but rapidly shrink as time runs out. When the player's remaining time reaches 0, the time-bar will be gone completely.

At the same time as adding the time-bar, we will add the necessary code to keep track of the player's remaining time, as well as respond when they run out. Let's go through it step by step.

Find the `Clock clock;` declaration from earlier and add the highlighted code just after, as shown next.

```
// Variables to control time itself
Clock clock;
// Time bar
RectangleShape timeBar;
float timeBarStartWidth = 400;
float timeBarHeight = 80;
timeBar.setSize(Vector2f(timeBarStartWidth, timeBarHeight));
timeBar.setFillColor(Color::Red);
timeBar.setPosition((1920 / 2) - timeBarStartWidth / 2, 980);
Time gameTimeTotal;
float timeRemaining = 6.0f;
```

```
float timeBarWidthPerSecond = timeBarStartWidth / timeRemaining;
// Track whether the game is running
bool paused = true;
```

First, we declare an object of type `RectangleShape` and call it `timeBar.` `RectagleShape` is an SFML class that is perfect for drawing simple rectangles.

Next, we add a couple of `float` variables, `timeBarStartWidth` and `timeBarHeight`. We initialize them to `400` and `80`, respectively. These variables will help us keep track of the size we need to draw `timeBar` each frame.

Next, we set the size of `timeBar` using the `timeBar.setSize` function. We don't just pass in our two new `float` variables. First, we create a new object of type `Vector2f`. What is different here, however, is that we don't give the new object a name. We simply initialize it with our two `float` variables and it is passed straight into the `setSize` function.

`Vector2f` is a class that holds two `float` variables. It also has some other functionality that will be introduced in the book.

After that, we color `timeBar` red by using the `setFillColor` function.

The last thing we do to `timeBar` in the previous code is to set its position. The vertical coordinate is completely straightforward, but the way we set the horizontal coordinate is slightly convoluted. Here is the calculation again:

```
(1920 / 2) - timeBarStartWidth / 2
```

The code first divides `1920` by `2`. Then it divides `timeBarStartWidth` by 2. Finally, it subtracts the latter from the former.

The result makes `timeBar` sit neatly and centrally horizontally on the screen.

The final three lines of code that we are talking about declare a new `Time` object called `gameTimeTotal`, a new `float` called `timeRemaining` that is initialized to `6`, and a curious-sounding `float` named `timeBarWidthPerSecond`, which we will discuss further next.

The `timeBarWidthPerSecond` variable is initialized with `timeBarStartWidth` divided by `timeRemaining`. The result is exactly the amount of pixels that `timeBar` needs to shrink by, each second of the game. This will be useful when we resize `timeBar` in each frame of the game loop.

Obviously, we need to reset the time remaining each time the player starts a new game. The logical way to do this is the *Enter* key press. We can also set `score` back to `0` at the same time. Let's do that now by adding this highlighted code.

```
// Start the game
if (Keyboard::isKeyPressed(Keyboard::Return))
{
paused = false;
// Reset the time and the score
score = 0;
timeRemaining = 6;
}
```

Now, at each frame, we must reduce the amount of time remaining and resize `timeBar` accordingly. Add the following highlighted code in the update section as shown here.

```
/*
****************************************
Update the scene
****************************************
*/
if (!paused)
{
// Measure time
Time dt = clock.restart();
// Subtract from the amount of time remaining
timeRemaining -= dt.asSeconds();
// size up the time bar
timeBar.setSize(Vector2f(timeBarWidthPerSecond *
timeRemaining, timeBarHeight));
// Set up the bee
if (!beeActive)
{
// How fast is the bee
srand((int)time(0) * 10);
beeSpeed = (rand() % 200) + 200;
// How high is the bee
srand((int)time(0) * 10);
float height = (rand() % 1350) + 500;
spriteBee.setPosition(2000, height);
beeActive = true;
}
else
// Move the bee
```

In the preceding code, first we subtracted the amount of time the player has left by however long the previous frame took to execute with this code.

```
timeRemaining -= dt.asSeconds();
```

Then we adjusted the size of `timeBar` with the following code:

```
timeBar.setSize(Vector2f(timeBarWidthPerSecond *
```

```
    timeRemaining, timeBarHeight));
```

The `x` value of `Vector2F` is initialized with `timebarWidthPerSecond` multiplied by `timeRemaining`. This produces exactly the correct width, relative to how long the player has left. The height remains the same and `timeBarHeight` is used without any manipulation.

And, of course, we must detect when time has run out. For now, we will simply detect that time has run out, pause the game, and change the text of `messageText`. Later, we will do more work here. Add the highlighted code right after the previous code we added and we will look at it in more detail.

```
// Measure time
Time dt = clock.restart();
// Subtract from the amount of time remaining
timeRemaining -= dt.asSeconds();
// resize up the time bar
timeBar.setSize(Vector2f(timeBarWidthPerSecond *
timeRemaining, timeBarHeight));
if (timeRemaining <= 0.0f) {
// Pause the game
paused = true;
// Change the message shown to the player
messageText.setString("Out of time!!");
//Reposition the text based on its new size
FloatRect textRect = messageText.getLocalBounds();
messageText.setOrigin(textRect.left +
textRect.width / 2.0f,
textRect.top +
textRect.height / 2.0f);
messageText.setPosition(1920 / 2.0f, 1080 / 2.0f);
}
// Set up the bee
if (!beeActive)
{
// How fast is the bee
srand((int)time(0) * 10);
beeSpeed = (rand() % 200) + 200;
```

```
// How high is the bee
srand((int)time(0) * 10);
float height = (rand() % 1350) + 500;
spriteBee.setPosition(2000, height);
beeActive = true;
}
else
// Move the bee
```

Stepping through the previous code:

- First we test whether time has run out with `if(timeRemaining <= 0.0f)`.
- Then we set `paused` to `true` so this will be the last time the update part of our code is executed (until the player presses *Enter* again).
- Then we change the message of `messageText`, calculate its new center to set as its origin, and position it in the center of the screen.

Finally, for this part of the code, we need to draw `timeBar`. There is nothing new in this code that we haven't seen many times before. Just note that we draw `timeBar` after the tree, so it is not partially obscured. Add the highlighted code to draw the time-bar.

```
// Draw the score
window.draw(scoreText);
// Draw the timebar
window.draw(timeBar);
if (paused)
{
// Draw our message
window.draw(messageText);
}
// Show everything we just drew
window.display();
```

Now you can run the game, press *Enter* to start, and watch the time-bar smoothly disappear down to nothing.

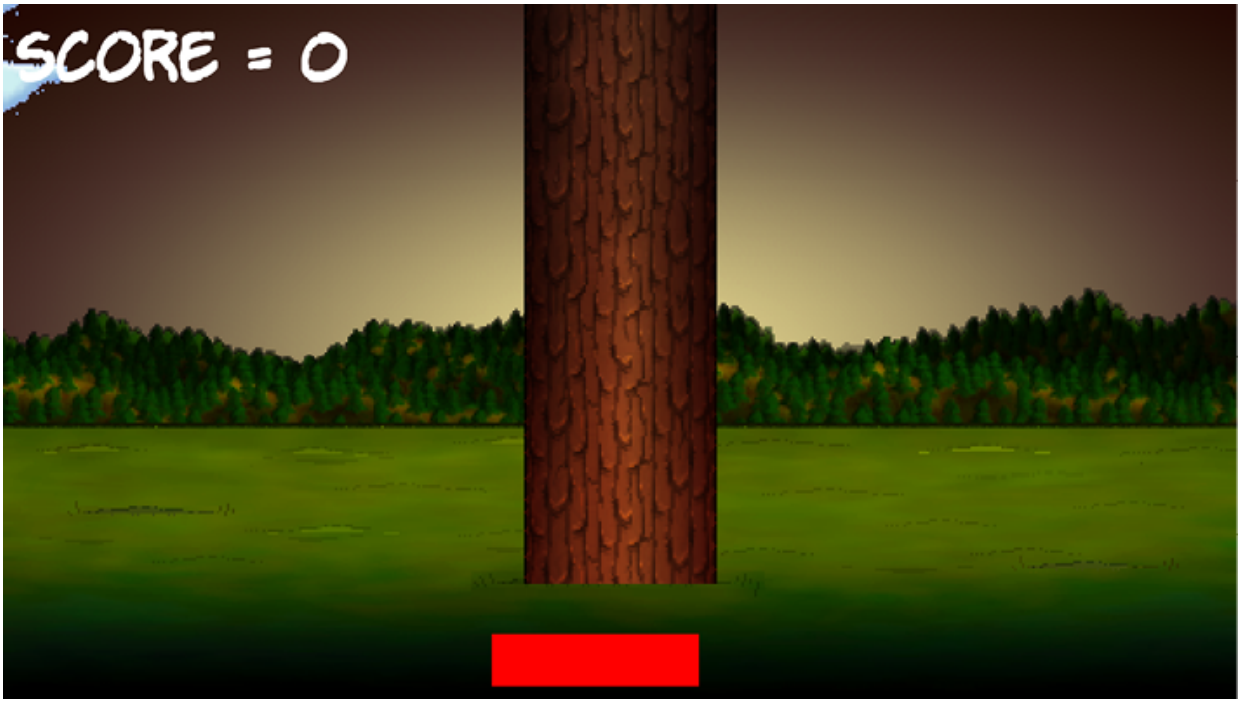


Figure 3.3: Time-bar disappearing

The game then pauses and the **OUT OF TIME!!** message will appear neatly in the center of the screen.

Figure 3.4: Time over

You can, of course, press *Enter* again to have the whole thing run from the start.

Summary

In this chapter, we learned about strings, SFML `Text`, and SFML `Font`. Between them, they enabled us to draw text to the screen, which provided the player with a HUD. We also used `sstream`, which allows us to concatenate strings and other variables to display the score.

We explored the SFML `RectangleShape` class, which does exactly what its name suggests. We used an object of type `RectangleShape` and some carefully planned variables to draw a time-bar that displays to the player how much time they have left. Once we have implemented chopping and moving branches that can squash the player, the time-bar will create tension and urgency.

Next, we are going to learn about a whole range of new C++ features, including loops, arrays, switching, enumerations, and functions. This will enable us to move the tree branches, keep track of their locations, and squash the player.

Frequently asked questions

Q) I can foresee that positioning sprites by their top-left corner could sometimes be inconvenient. Is there an alternative?

A) Fortunately, you can choose what point of a sprite is used as the positioning/origin pixel, just like we did with `messageText`, using the `setOrigin` function.

Q) The code is getting rather long and I am struggling to keep track of where everything is. How can we fix this?

A) Yes, I agree. In the next chapter, we will look at the first of a few ways we can organize our code and make it more readable. We will see this when we learn about writing C++ functions. In addition, we will learn a new way of handling multiple objects/variables of the same type (like the clouds) when we learn about C++ arrays.

Q) I couldn't get my font to load. How do I know what is going on behind the scenes? How do I know if I have entered the correct file path or mistyped the name of the font file?

A) We can wrap our font-loading code in an `if` statement and include some error-handling code using `cout` as well. Here is an example:

```
if (!font.loadFromFile("arial.ttf")) {
        // If the loading fails, display an error message
```

```
        cout << "Error loading font!";
}
```

Now, if the font doesn't load, the execution will continue with missing text but you will get an error message printed to the console to inform you. You can do the same with loading textures as well, as this code shows:

```
if (!texture.loadFromFile("texture.png")) {
        // If the loading fails, display an error message
        cout << "Error loading texture!";
    }
```

4

Loops, Arrays, Switch, Enumerations, and Functions: Implementing Game Mechanics

This chapter probably has more C++ information than any other chapter in the book. It is packed with fundamental concepts that will accelerate our understanding enormously. It will also begin to shed light on some of the murky areas we have been skipping over a little bit, like functions, the game loop, and loops in general.

This is what we will explore:

- Loops
- Arrays
- Making decisions with switch
- Class enumerations
- Getting started with functions
- Growing the branches

Once we have explored a whole list of C++ language necessities, we will then use everything we know to make the main game mechanic—the tree branches—move. By the end of this chapter, we will be ready for the final phase and the completion of Timber!!!.

Loops

Welcome to the world of loops in C++! **Loops** are general programming constructs not unique to C++ that allow you to repeat a certain block of code multiple times. They are crucial for making our games more efficient and flexible. It is probably the vital thing that makes computers so useful: doing the same thing but with different values repeatedly. In C++, there are several types of loops, each serving specific purposes. In this chapter, we'll explore the fundamental loop structures and cover relatively recent updates to C++ that affect your loop-related programming options. The obvious example that we have seen so far is the game loop. With all the code stripped out, our game loop looks like this.

```
while (window.isOpen())
{
}
```

The correct term for this type of loop is a **while** loop. Let's look at that first.

while loops

The `while` loop is quite straightforward. Think back to the `if` statements and their expressions that evaluated to either `true` or `false`. We can use the exact same combination of operators and variables in the conditional expression of our `while` loops.

As with `if` statements, if the expression is `true`, the code executes. The difference with a `while` loop, however, is that the C++ code within it will repeatedly execute, potentially forever, until the condition is `false`. Look at this code:

```
int numberOfZombies = 100;
while(numberOfZombies > 0)
{
  // Player kills a zombie
  numberOfZombies--;
  // numberOfZombies decreases each pass through the loop
}
// numberOfZOmbies is no longer greater than 0
```

This is what happened in the previous code. Outside of the `while` loop, `int numberOfZombies` is declared and initialized to `100`. Then, the `while` loop begins. Its conditional expression is `numberOfZombies > 0`. Consequently, the `while` loop will continue looping through the code in its body until the condition evaluates to `false`. This means that the code above will be executed 100 times.

On the first pass through the loop, `numberOfZombies` equals 100, then 99, then 98, and so on. But once `numberOfZOmbies` is equal to zero, it is, of course, no longer *greater* than zero. Then, the code will break out of the `while` loop and continue to run, after the closing curly brace.

Just like an `if` statement, it is possible that the `while` loop will not execute even once. Look at this next code:

```
int availableCoins = 10;
while(availableCoins > 10)
{
  // more code here.
// Won't run unless availableCoins is greater than 10
}
```

In the preceding code, the loop condition evaluates to `false` because `availableCoins` is not greater than 10. As the condition is `false`, the loop does not execute even once.

Moreover, there is no limit to the complexity of the expression or the amount of code that can go in the loop body. We have already put quite a lot of code in our game loop. Consider this hypothetical variation of a game loop:

```
int playerLives = 3;
int alienShips = 10;
while(playerLives !=0 && alienShips !=0 )
{
   // Handle input
// Update the scene
// Draw the scene
}
// continue here when either playerLives or alienShips equals 0
```

The previous `while` loop would continue to execute until either `playerLives` or `alienShips` was equal to zero. As soon as one of those conditions occurred, the expression would evaluate to `false` and the program would continue to execute from the first line of code after the `while` loop.

It is worth noting that once the body of the loop has been entered, it will try to complete at least once, even if the expression evaluates to `false` part way through, as the condition is not tested again until the code tries to start another pass. As an example, look at this code:

```
int x = 1;
while(x > 0)
{
  x--;
  // x is now 0 so the condition is false
// But this line still runs
// and this one
// and me!
}
// Now I'm done!
```

The previous loop body will execute once. We can also set up a `while` loop that will run forever, appropriately called an **infinite loop**. Here is an example:

```
int y = 0;
while(true)
{
   y++; // Bigger... Bigger...
   cout << y;
}
```

If you find the above loop confusing, just think of it literally. A loop executes when its condition is `true`. Well, `true` is always `true` and will therefore keep executing. The value of `y` will be printed each time through the loop as it increases by one on each pass.

As an interesting aside, there is a limit to how big y will get. If you check the table of variable types back in *Chapter 2*, you will notice that an `int` holds a maximum size. An `int` can vary from 32- or 64-bit machines and even the brand of compiler can affect the values an `int` holds, but typically, an `int` is 16 bits of data and can represent -32,767 to 32,767. The preceding code would add up to the maximum 32,767, then the next value would be -32,767, and then 32,767 iterations of the loop later, `y` will be back to zero. You can try this out by creating an empty console app and pasting the preceding code in the `main` function. None of the complicated SFML configurations are necessary, just remember to put `#include <iostream>` at the top of your

> code and `using namespace std;` before the main function to be able to use `cout`.

Whether a loop is infinite or not, we sometimes need a way to break out of the loop earlier than the loop condition allows. For example, a game loop that tracks if the player or the aliens are all dead is fine, but what if the player just wants to quit early? Here is how to do it.

Breaking out of a loop

We might use an infinite loop so that we can decide when to exit the loop from within its body rather than in the expression. We would do this by using the `break` keyword when we are ready to leave the loop body, perhaps like this:

```
int z = 0;
while(true)
{
  z++; // Bigger... Bigger...
  cout << z;
 break; // No you're not
// Code doesn't reach here
}
```

In the preceding code, `z` first equals zero, then it is incremented with `z++`, and then the value of `z` is printed with `cout`. Immediately after, however, the `break` keyword makes the code exit the loop. The `break` keyword has this effect even if there are more lines of code that follow it. What is potentially even more useful is we can *conditionally* use `break`, as we discuss next.

You might also have been able to guess that we can combine any of the C++ decision-making tools (like `if`, `else`, and another we will learn shortly,

`switch`) within our `while` loops and other loop types as well. Consider this example:

```
int x = 0;
int max = 10;
while(true)// Potentially infinite
{
  x++; // Bigger... Bigger...
 if(x == max)// Not infinite anymore
{
 break;
  }
// code reaches here only until max equals 10
}
```

This code demonstrates a controlled use of an infinite loop that is exited based on a specific condition (`x == max`). It is used when you need to perform a task repeatedly until a certain condition is met. In this case, it increments `x` until it reaches the value of `max`, at which point, the loop is exited.

As a final example of `while` loops, let's look at how the user can determine when a `while` loop exits. Of course, we, as the game programmers, will determine the format and timing of the player's choices. In this next example, I also introduce a new keyword, `cin`. See if you can work out what is happening:

```
int userInput;
while (true)
{
    cout << "Enter a positive number to exit: ";
    cin >> userInput;
    if (userInput > 0)
  {
        break;
```

```
        }
        cout << "Invalid input. Try again.";
    }
```

This example uses a `while` loop for validating user input. The loop continues until the user enters a positive number, using `break` to exit the loop when the condition is met.

The user input is achieved using `cin`, which pauses execution and waits for the user to enter a number and then press the *Return* key. Notice the operator used with `cin` points the other way, `>>` instead of `<<`. This operator is called the extraction operator.

The code continuously prompts the user, and a `break` statement exits the loop when a valid (greater than zero) input is received.

As a final word on using the `break` keyword, it is generally considered good practice to use it sparingly as it can make the code harder to understand. Don't be afraid of using it; there are definitely times when it is exactly what you need. Sometimes, while trying to think about the best form for a loop, I find that I have forgotten about `break`, and then it comes back to me and I realize it is just what I need. A good rule of thumb is not to *try* and design in `break` from the start but accept it as a valid solution if it presents itself as such and a clearer solution is not apparent.

If you want to try out the preceding, copy it into the `main` function of an existing or new console app. None of the complicated SFML configurations are necessary, just remember to put `#include <iostream>` at the top of your

code and `using namespace std;` before the `main` function to be able to use `cout` and `cin`.

To dig a bit deeper with `cin`, it is an object that facilitates the reading of user inputs from the console. Paired with the **extraction operator** `>>`, `cin` allows us to acquire inputs interactively during program execution. If you wanted to write a text adventure 1970s/80s style `cin`, `cout`, loops, variables, and conditions would be almost all you need. cin is an instance of a class; it is an object. Somebody else programmed the class, in this case, the `istream` class, and we created an instance of it with `cin` and used a super-useful feature without worrying about how it works. This class/instance/object conundrum will make perfect sense when we discuss it properly in *Chapter 6*.

We could go on for a long time looking at the various permutations of C++ `while` loops, but at some point, we want to get back to making games. So let's move on to another type of loop.

for loops

`for` loops in C++ are designed for when we need to iterate over a range of values. They provide a concise way to execute a set of statements repeatedly.

A typical `for` loop consists of three parts: initialization, condition, and iteration statement, making it easy to control the loop's execution. `for` loops are especially useful when the number of iterations is known in advance.

It is because of the three parts that the `for` loop has a slightly more complicated syntax than a `while` loop because it takes three parts to set one up. Have a look at the code first and then we will break it apart:

```
for(int x = 0; x < 100; x ++)
{
```

```
    // Something that needs to happen 100 times goes here
}
```

Here is what all the parts of the `for` loop condition do:

`for`(declaration and initialization`;` condition`;` change before each iteration)

To clarify further, here is a table to explain each of the three key parts as they appear in the previous `for` loop example.

Part	**Description**
Declaration and initialization	We create a new `int` variable `i` and initialize it to `0`
Condition	Just like the other loops, it refers to the condition that must be `true` for the loop to execute
Change after each pass through the loop	In the example, `x ++` means that 1 is added/incremented to `x` on each pass

Table 4.1: Key parts for loop

In summary, the preceding `for` loop code utilizes the loop to iterate 100 times. It initializes a loop variable `x` to zero, sets the loop condition to continue as long as `x` is less than 100, and increments `x` by 1 in each iteration. The block of code inside the loop, indicated by the curly braces, represents the job to be performed 100 times. This is useful when you have code that should be executed repeatedly a predetermined number of times. In this case, the loop allows for concise and clear code for handling a repetitive task.

We can vary `for` loops to do many more things. Here is another simple example that counts down from 10:

```
for(int i = 10; i > 0; i--)
{
  // countdown
}
// blast off
```

The `for` loop takes control of **initialization**, **condition evaluation**, and the **control variable** itself. We will use `for` loops in our game, later in this chapter. `for` loops have more advanced uses too, but we need to learn about some more topics to be able to discuss them. We will see one of these more advanced uses in the next section when we talk about arrays.

Arrays

Arrays are **data structures** that allow us to store collections of elements of the same data type using a single name, perhaps `someInts`, `myFloats`, or `zombieHorde`. Arrays provide a convenient way to organize and manipulate data, enabling more efficient and structured programming. Arrays are especially useful for repetitive data, like lists of numbers, characters, or game objects. This introduction will explore the basics of arrays and, as we proceed through the book, we will see more advanced uses.

A comparison with a regular variable might help. If a variable is a box in which we can store a value of a specific type, like `int`, `float`, or `char`, then we can think of an array as a row of boxes. The row of boxes can be of almost any size and type, including objects made from classes. However, all the boxes must be of the same type.

The limitation of having to use the same type in each box can be circumvented to an extent once we learn some more advanced C++ in the final platformer project.

If you think this array sounds like it could have been useful for our clouds from *Chapter 2*, *Variables, Operators, and Decisions: Animating Sprites*, you are exactly right. It is too late for the clouds, they are destined to be clunky bloated code forever. The tree branches, however, we will implement using arrays. So how do we go about creating and using an array?

Declaring an array

We can declare an array of `int` type variables like this:

```
int someInts[10];
```

Now we have an array called `someInts` that can store 10 `int` values. Currently, however, it is empty.

The only difference with regular variables is that we would use a format known as **array notation** to manipulate the individual values as, although our array has a name—`someInts`—the individual elements do not have individual names:

```
someInts_AliensRemaining = 99; // Wrong
someInts_Score = 100; // Wrong!
```

Let's see exactly how we do it.

Initializing the elements of an array

To add values to the elements of an array, we can use the type of syntax we are already familiar with combined with the new syntax I mentioned, known as **array notation**. In this next code, we store the value of `99` in the first **element** of the array:

```
someInts[0] = 99;
```

To store a value of 999 in the second element, we write this code:

```
someInts[1] = 999;
```

We can store a value of 3 in the last element like this:

```
someInts[9] = 3;
```

Note that the elements of an array always start at zero and go up to the size of the array minus one. Similar to ordinary variables, we can manipulate the values stored in an array.

In this next code, we will see how we manipulate the individual values. This is how we add the first and second elements together and store the answer in the third:

```
someInts[2] = someInts[0] + someInts[1];
```

Arrays can also interact seamlessly with regular variables, like this perhaps:

```
int a = 9999;
```

```
someInts[4] = a;
```

There is much to learn about arrays, so let's keep going.

Quickly initializing the elements of an array

We can quickly add values to the elements like this example, which uses a `float` array:

```
float myFloatingPointArray[3] {3.14f, 1.63f, 99.0f};
```

Now the values `3.14`, `1.63`, and `99.0` are stored in the first, second and third positions, respectively. Remember that when using array notation to access these values, we would use [0], [1], and [2].

There are other ways to initialize the elements of an array. This slightly abstract example shows using a `for` loop to put the values 0 through 9 into the `uselessArray` array:

```
for(int i = 0; i < 10; i++)
{
  uselessArray[i] = i;
}
```

The code assumes that `uselessArray` had previously been initialized to hold at least 10 `int` variables.

What do these arrays really do for our games?

We can use arrays anywhere a regular variable can be used, perhaps in an expression like this:

```
// someArray[4] is declared and initialized to 9999
for(int i = 0; i < someArray[4]; i++)
{
  // Loop executes 9999 times
}
```

Perhaps the biggest benefit of arrays in game code was hinted at at the start of this section. Arrays can hold objects (instances of classes). Imagine that we have a `Zombie` class, and we want to store a whole bunch of them. We could do so like this hypothetical code:

```
Zombie horde [5] {zombie1, zombie2, zombie3}; // etc...
```

The `horde` array now holds a load of instances of the `Zombie` class. Each one is a separate, living (kind of), breathing, self-determining `Zombie` object. We could then loop through the `horde` array, each pass through the game loop, moving the zombies, and checking if their heads have met with an axe or if they have managed to catch the player.

Had we known about them at the time, arrays would have been perfect for handling our clouds. We could have had hundreds of clouds and written much less code than we did for our three measly clouds.

To check out this improved cloud code in full and in action, look at the enhanced version of Timber!!! in the download bundle in the *Chapter 5* folder. Or you can try to implement the clouds using arrays yourself before looking at the code.

The best way to get a feel for all this array stuff is to see it in action. And we will when we implement our tree branches.

For now, we will leave our cloud code as it is so we can get back to adding features to the game as soon as possible. But first, a bit more C++ decision-making with **switch**.

Making decisions with switch

We have already seen the `if` keyword, which allows us to decide whether to execute a block of code based on the result of its expression, but sometimes, a decision in C++ can be better made in other ways. It is often used for providing an elegant alternative to a series of nested `if-else` statements. As we will see, it evaluates an expression and directs program flow.

When we must make a decision based on a clear list of possible outcomes that don't involve complex combinations or wide ranges of values, then `switch` is usually the way to go. We start a `switch` decision like this:

```
switch(expression)
{
  // More code here
}
```

In the previous example, `expression` could be an actual expression or just a variable. Then, within the curly braces, we can make decisions based on the result of the expression or value of the variable. We do this with the `case` and `break` keywords, as in this slightly abstract example:

```
case x:
    //code for x
break;
```

```
case y:
    //code for y
break;
```

You can see, in the previous abstract example, that each `case` states a possible result, and each `break` denotes the end of that `case` and the point that execution leaves the `switch` block.

The classic, non-abstract example is using days of the week, as shown next:

```
int dayNumber = 3;
switch (dayNumber)
{
    case 1:
        // what happens on Monday
break;
    case 2:
        // what happens on Tuesday
break;
      // etc
default:
        // code for an invalid day
}
```

In the preceding code, an `int` variable called `dayNumber` is given the value 3, representing a day of the week. The `switch` condition evaluates the value of `dayNumber`. Each case corresponds to a specific day, with a block of code for each.

However, something new has been introduced. We can also, optionally, use the `default` keyword without a value, to run some code in case none of the `case` statements evaluate to `true`. This is a bit like the `else` keyword without an expression following an `if` expression, perhaps like this code:

```
default: // Look no value
// Do something here if no other case statements are true
break;
```

As a final example for `switch`, consider a retro text adventure where the player enters a letter like ‘n’, ‘e’, ‘s’, or ‘w’ to move north, east, south, or west. A `switch` block could be used to handle each possible input from the player:

```
// get input from user in a char called command
char command;
cin >> command;
switch(command){
  case 'n':
    // Handle move here
break;
  case 'e':
    // Handle move here
break;
  case 's':
    // Handle move here
break;
  case 'w':
    // Handle move here
break;

  // more possible cases
default:
    // Ask the player to try again
break;
}
```

The best way of understanding all we have seen regarding `switch` will be when we put it into action along with all the other new concepts we are learning. First, we need to understand enumerations, which help us be more precise in our code.

Class enumerations

An enumeration is a list of all the possible values in a logical collection. C++ enumerations are a great way of, well, enumerating things. For example, if our game uses variables that can only be in a specific range of values, and if those values could logically form a collection or a set, then enumerations are probably appropriate to use. They will make your code clearer and less error-prone. For example, in the `switch` example using days of the week, who gets to decide what the first day of the week is? And what if somebody thinks that `dayNumber` is something else and does some arithmetic on it? All of a sudden, our day numbering system is a mess. Class enumerations solve this and other problems.

To declare a class enumeration in C++, we use the two keywords `enum class` together, followed by the name of the enumeration, followed by the values the enumeration can contain, enclosed in a pair of curly braces, `{...}`.

As an example, examine this enumeration declaration. Note that it is conventional to declare the possible values from the enumeration in all uppercase:

```
enum class daysOfWeek {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRI
```

Or a more interesting example in a possible game scenario like this:

```
enum class zombieTypes {REGULAR, RUNNER, CRAWLER, SPITTER, BLOAT
```

Note that, at this point, we have not declared any instances of `zombieType`, just the structure and metadata of the type itself. If that sounds odd, think about it like this. SFML created the `Sprite`, `RectangleShape`, and `RenderWindow` classes, but to use any of those classes, we had to declare an object/instance of the class.

At this point, we have created a new type called `zombieTypes`, but we have no instances of it. So, let's do that now:

```
zombieType Rishi = zombieTypes::CRAWLER;
zombieType Suella = zombieTypes::SPITTER
zombieType Boris = zombieTypes::BLOATER
/*
  Zombies are fictional creatures and any resemblance
  to real people is entirely coincidental
*/
```

Next is a sneak preview of the type of code we will soon be adding to Timber!!!. We will want to track which side of the tree a branch or the player is on, so we will declare an enumeration called `side`, like this:

```
enum class side { LEFT, RIGHT, NONE };
```

We could position the player on the left like this:

```
// The player starts on the left
side playerSide = side::LEFT;
```

We could make the fourth level (arrays start from zero) of an array of branch positions have no branch at all, like this:

```
branchPositions[3] = side::NONE;
```

It is good to know that we can use enumerations in expressions as well:

```
if(branchPositions[5] == playerSide)
{
  // The lowest branch is the same side as the player
// SQUISHED!!
}
```

Furthermore, we can use enumerations with switch too. This next code clarifies our previous days of the week example:

```
daysOfWeek day = daysOfWeek::WEDNESDAY;
    switch (day) {
        case daysOfWeek::MONDAY:
            std::cout << "It's Monday";
            break;
        case daysOfWeek::TUESDAY:
            std::cout << "It's Tuesday";
            break;
        case daysOfWeek::WEDNESDAY:
            std::cout << "It's Wednesday";
            break;
        case daysOfWeek::THURSDAY:
            std::cout << "It's Thursday";
            break;
        case daysOfWeek::FRIDAY:
            std::cout << "It's Friday";
            break;
        case daysOfWeek::SATURDAY:
            std::cout << "It's Saturday";
            break;
        case daysOfWeek::SUNDAY:
            std::cout << "It's Sunday";
            break;
        default:
            std::cout << "OOPS try again.";
    }
```

In the preceding example, the `daysOfWeek` enumeration is used instead of `int`. The `switch` statement evaluates the day variable, and each case corresponds to a specific day of the week. As before, the `default` case handles any invalid day that might be encountered. In the preceding example, it is totally clear that the code block for Wednesday will execute.

We will look at one more vital C++ topic and then we will get back to coding the game.

Getting started with functions

Welcome to the world of C++ functions. **Functions** are one of the fundamental building blocks of C++ programming. When I said earlier that you probably knew enough C++ to write a retro text adventure, after learning about functions, you definitely will!

Functions allow us to wrap reusable parts of our code and create well-organized programs. The rest of this chapter will walk you through the essentials of functions, from their basic syntax to more advanced concepts, providing you with a comprehensive foundation and ending on how functions and classes are part of the same topic. You will then be ready to finish this game in this chapter and the next and stride confidently on to *Chapter 6*, where we will finally tackle the topic of object-oriented programming.

What exactly are C++ functions? A function is a collection of variables, expressions, and **control flow statements** (loops and branches). In fact, any of the code we have learned about in the book so far can be used in a function. In fact, all the code we have written so far has been in the `main` function. A quick glance at our project code so far will show that we have

hundreds of lines of code. As suggested in the introduction to functions, we will soon begin to separate (modularize) and organize (encapsulate) all future code into manageable chunks.

I considered the idea of rewriting Timber!!! as we learned better ways of doing things but decided it would be better to leave that as an exercise for those of you who wanted to do so.

The first part of a function that we write is called the **signature**. Here is an example function signature:

```
public void shootLazers(int power, int direction)
```

If we add an opening and closing pair of curly braces, `{...}`, with some code that the function performs, we then have a complete function, a **definition**:

```
public void shootLazers(int power, int direction)
{
    // ZAPP!
}
```

We could then use our new function from another part of our code, perhaps like this:

```
// Attack the player
shootLazers(50, 180) // Run the code in the function
// I'm back again - code continues here after the function ends
```

When we use a function, we say that we **call** it. At the point where we call `shootLazers`, our program's execution branches to the code contained within that function. The function would run until it reaches the end or is told to

`return`. Then, the code would continue running from the first line after the function call. We have already been using the functions that SFML provides. What is different here is that we will learn to write and call our *own* functions.

Here is another example of a function, complete with the code to make the function return to the code that called it:

```
int addAToB(int a, int b)
{
  int answer = a + b;
  return answer;
}
```

The call to use the above function could look like this:

```
int myAnswer = addAToB(2, 4);
```

Obviously, we don't need to write functions to add two variables together, but the overly simplified example helps us see a little more into the workings of functions. First, we pass in values `2` and `4`. In the function signature, the value `2` is assigned to `int a` and the value `4` is assigned to `int b`.

Within the function body, the variables `a` and `b` are added together and used to initialize the new variable `int answer`. The line `return answer;` does just that. It returns the value stored in `answer` to the calling code, causing `myAnswer` to be initialized with the value `6`.

Notice that each of the function signatures in the examples above varies a little. The reason for this is that the C++ function signature is quite flexible, allowing us to build exactly the functions we require.

Exactly how the function signature defines how the function must be called and if/how the function must return a value deserves further discussion. Let's give each part of that signature a name so we can break it into parts and learn about them.

Here is a function signature with its parts described by their formal/technical term:

return type | **name of function** | (**parameters**)

Here are a few examples we can use for each of those parts:

- **Return-type**: `bool`, `float`, `int`, etc., or any C++ type or expression.
- **Name of function**: `shootLazers`, `addAToB`, etc.
- **Parameters**: `(int number, bool hitDetected)`, `(int x, int y)`, `(float a, float b)`

At this point, a brief interlude into the design of C++, programming, and computer hardware might be worthwhile.

Who designed all this weird and frustrating syntax and why is it the way it is?

Sometimes, beginners to C++ will question the way the language is designed, and functions is a topic (as well as OOP) in particular when the syntax enforced upon us as developers is questioned for its design. The point to remember is that the syntax of C++ and functions in particular weren't just designed in a vacuum. They were designed and chosen around the way that a computer system (in particular, a CPU) works.

As we have learned, in C++, functions help us organize and modularize our code. When a function is called, several steps occur.

As we know, when a function is called, the program's control flow transfers to the function. The CPU executes a jump instruction to the memory address associated with the function. This memory address is hidden from us but, actually, it is contained in the function's name that we assign to it.

Next, a stage called the **function prologue** is executed, which involves setting up the function's **stack frame**. This is completely hidden from us as programmers, but it is part of how the CPU handles things. The current state of the calling function, often `main`, is stored, including the return address and the values of important CPU registers that hold values.

At this stage, our variables and function parameters are allocated on the **stack**. The stack is a region of computer memory internal to the CPU used for the dynamic storage of function call information, local variables, and control flow data. Our function parameters are typically passed through CPU registers or pushed onto this stack. Variables within the function itself, known as **local variables**, are created on the stack and initialized.

Next, the body of the called function executes and local variables and parameters are accessed within the function.

Before returning from the function, the **function epilogue** is executed. The function epilogue is the set of instructions executed before returning from a function, typically involving the deallocation of the function's stack frame and the restoration of the saved state of the calling function. The stack frame is deallocated, freeing up space for local variables and parameters. The saved state of the calling function is restored, including the return address.

After the epilogue, the CPU executes a return instruction, transferring control back to the calling function. The return value from the function (if any) is stored in a pre-determined register.

The stack pointer is a register that keeps track of the top of the stack. During function calls, the stack pointer is adjusted to allocate and deallocate space for local variables and parameters. This is important because you can call a function, which calls another function, and so on. In fact, most complex applications, including games, will have many functions on the stack.

The stack follows a **Last In, First Out** (**LIFO**) order, meaning the last item pushed onto the stack is the first to be popped off. This is why it is called a stack. The best analogy I have heard to visualize the stack is that of a stack of plates at a buffet where the plates are constrained in a device to make only the top plate accessible. The restaurant manager can always add to the stack by pushing new plates onto the spring-loaded device, but to get to the plate at the bottom of the stack, each plate must be individually removed.

In summary, when a function is called, the CPU uses the stack to manage the function's local variables and the called function parameters. The stack pointer keeps track of the stack's top, and the function prologue and epilogue handle the setup and cleanup of the stack. This process allows for the efficient execution of multiple nested function calls. Understanding the interaction between functions and the CPU, hopefully, helps us appreciate half a century of refinement and improvement into the state of C++ today and not be too critical of the syntax we are forced to learn. It is like it is for a reason.

It is not necessary to understand how a CPU works, not even the above brief introduction, but knowing that C++ is the culmination of half a century of very careful and deliberate evolution from the early 1970s when the C

programming language was being developed can help beginners to accept that there probably isn't a "better" way and to embrace all the apparent imperfections as a necessity for taking efficient control of the great wonder of modernity, the CPU. Over time, if you stick at it, it will all become obvious why it was done how it was and, while it isn't necessary, a knowledge of computer hardware like the CPU and the GPU is useful to aid in understanding.

Now, with karma restored, let's look at each part of a function in turn.

Function return types

The return type, as the name suggests, is the type of the value that will be returned from the function to the calling code:

```
int addAToB(int a, int b){
    int answer = a + b;
    return answer;
}
```

In our slightly dull but useful `addAtoB` example previously, the return type in the signature is `int`. The function `addAToB` sends back (returns) to the code that called it, a value that will fit in an `int` variable. The return type can be any C++ type we have seen so far or one of the ones we haven't seen yet.

A function does not have to return a value at all, however. In this case, the signature must use the `void` keyword as the return type. When the `void` keyword is used, the function body must not attempt to return a value as this will cause an error. It can, however, use the `return` keyword without a value. Here are some combinations of return type and use of the `return` keyword that are valid:

```
void doWhatever(){
    // our code
// I'm done going back to calling code here
// no return is necessary
}
```

Another possibility is as follows:

```
void doSomethigCool(){
  // our code
// I can do this if I don't try and use a value
return;
}
```

The following code shows yet more examples of possible functions. Be sure to read the comments as well as the code:

```
void doYetAnotherThing(){
  // some code
if(someCondition){
    // if someCondition is true returning to calling code
// before the end of the function body
return;
  }
    // More code that might or might not get executed
return;
  // As I'm at the bottom of the function body
// and the return type is void, I'm
// really not necessary but I suppose I make it
// clear that the function is over.
}
bool detectCollision(Ship a, Ship b){
  // Detect if collision has occurred
if(collision)
  {
    // Bam!!!
return true;
  }
```

```
  else
  {
    // Missed
return false;
  }
}
```

The last function example above, `detectCollision`, is a glimpse into the near future of our C++ code and demonstrates that we can also pass in the user-defined types, called objects, into functions to perform calculations on them.

We could call each of the functions above in turn, like this:

```
// OK time to call some functions
doWhatever();
doSomethingCool();
doYetAnotherThing();
if (detectCollision(milleniumFalcon, lukesXWing))
{
  // The jedi are doomed!
// But there is always Leia.
// Unless she was on the Falcon?
}
else
{
  // Live to fight another day
}
// Continue with code from here
```

Don't worry about the odd-looking syntax regarding the `detectCollision` function; we will see real code like this quite soon. Simply, we are using the return value (`true` or `false`) as the expression, directly in an `if` statement.

Furthermore, the functions could be stacked up on the CPU stack if the functions were recoded as follows. I have stripped some extraneous code

like the comments and highlighted the new parts. First is a hypothetical `main` function:

```
int main()
{
  // call doWhatever
doWhatever()
  return 0;
}
```

Here is a new version of `doWhatever`:

```
void doWhatever(){
  // Call doSomethingCool
doSomethingCool();
}
```

Here is the new version of `doSomethingCool`:

```
void doSomethigCool(){
  // Call doYetAnotherThing
doYetAnotherThing();
  return;
}
```

Here is the new version of `doYetAnotherThing`:

```
void doYetAnotherThing(){
  if(someCondition){
    return;
  }
  return;
}
```

In my above scenario, the `main` function calls `doWhatever`, which calls `doSomethingCool`, which calls `doYetAnotherThing`. At this point, all four functions, including `main`, will exist on the CPU's stack. When `doYetAnotherThing` completes and goes through its epilogue process, is removed from the stack, and control returns to `doSomethingCool`. Then, only three functions exist on the stack. When `doSomethingCool` has had its code executed, it too is removed, and so on until just `main` is on the stack and, of course, eventually `main` reaches its `return` statement and is removed from the stack and our program is no longer in memory.

As a quick aside, loops also go through a similar process to functions, so if a function contains a loop, it too will end up on the stack. Everything is executed last in, first out until a return statement is reached, the currently executing function is removed, and the calling function continues.

That is more than you need to know to make a great game so let's keep going.

Function names

The function name when we design our own function can be almost anything at all. But it is best to use words, usually verbs, that clearly explain what the function will do. For example, look at this function:

```
void functionarooniebоonie(int blibbityblob, float floppyfloatyt
{
   //code here
}
```

The above is perfectly legal and will work, but these next function names are much clearer:

```
void doSomeVerySpecificTask()
{
  //code here
}
int getMySpaceShipHealth()
{
   //code here
}
void startNewGame()
{
   //code here
}
```

Next, take a closer look at how we share some values with a function.

Function parameters

We know that a function can return a result to the calling code. What if we need to share some data values from the calling code with the function? **Parameters** allow us to share values with the function. We have already seen examples of parameters while looking at return types. We will look at the same example but a little more closely:

```
int addAToB(int a, int b)
{
  int answer = a + b;
  return answer;
}
```

Above, the parameters are `int a` and `int b`. Did you notice that in the first line of the function body, we use `a + b` as if they are already declared and

initialized variables? Well, that's because they are. The parameters in the function signature are their declaration, and the code that calls the function initializes them.

Notice that we are referring to the variables in the function signatures brackets `(int a, int b)` as parameters. When we pass values into the function from the calling code, these values are called arguments. When the arguments arrive, they are used by the parameters to initialize real, usable variables: `int returnedAnswer = addAToB(10,5);`

Also, as we have partly seen in previous examples, we don't have to just use `int` in our parameters. We can use any C++ type. We can also use as many parameters as is necessary to solve our problem, but it is good practice to keep the parameter list as short and, therefore, manageable as possible.

As we will see in future chapters, we have left a few of the cooler uses of functions out of this introductory tutorial, so that we can learn about related C++ concepts before we take the topic of functions further.

The function body

The body is the part we have been kind of avoiding with comments like this:

```
// code here
// some code
```

But actually, we know exactly what to do here already! Any C++ code we have learned about so far will work in the body of a function.

Next, we will explore the concept of function prototypes.

Function prototypes

We have seen how to code a function and we have seen how to call one as well. There is one more thing we need to do, however, to make it work. All functions must have a **prototype**. A prototype is what makes the compiler aware of our function; without a prototype, the entire game will fail to compile. Fortunately, prototypes are straightforward.

We can simply repeat the function's signature, followed by a semicolon. The caveat is that the prototype must appear *before* any attempt to call or define the function. So, the absolute most simple example of a fully usable function in action is as follows. Look carefully at the comments and the location in the code where the different parts of the function appear:

```
// The prototype
// Notice the semicolon on the end
int addAToB(int a, int b);
int main()
{
  // Call the function
// Store the result in answer
int answer = addAToB(2,2);
  // Called before the definition
// but that's OK because of the prototype
// Exit main
return 0;
}// End of main
// The function definition
int addAToB(int a, int b)
{
    return a + b;
}
```

What the previous code demonstrates is the following:

- The prototype is before the `main` function.
- The call to use the function is, as we might expect, inside the `main` function.
- The definition is after/outside the `main` function.

Note that we can omit the function prototype and go straight to the definition when the definition occurs before the function is used. As our code becomes longer and spread across multiple files, however, this will almost never happen. We will use separate prototypes and definitions all the time.

Let's see how we can keep our functions organized.

Organizing functions

Well worth pointing out if we have multiple functions, especially if they are fairly long, is that our `.cpp` file will quickly become unwieldy. This defeats part of the objective that functions are intended for. The solution that we will see in the next project, which starts in *Chapter 6*, is that we can add all our function prototypes to our very own header file (`.hpp` or `.h`), then code all our functions in another `.cpp` file, and then simply add another `#include...` directive in our main `.cpp` file. This way, we can use any number of functions without adding any of their code (prototype or definition) to our main code file.

Function scope

We mentioned, in our discussion about the CPU stack, the idea of local variables. This is the same topic as function or variable **scope**. If we declare a variable in a function, either directly or in one of the parameters, that variable is not usable/visible outside of the function. Furthermore, any variables declared inside other functions cannot be seen/used inside the function. After all, they are in an entirely different stack frame on the CPU stack.

The way that we should share values between function code and calling code is through the parameters/arguments and the return value.

When a variable is not available because it is from another function, it is said to be out of scope. When it is available and usable, it is said to be in scope.

Variables declared within any block in C++ only are in scope within that block! This includes loops and `if` blocks as well. A variable declared at the top of `main` is in scope anywhere in `main`. A variable declared in the game loop is only in scope within the game loop, etc. A variable declared within a function or other block is called a **local** variable. The more code we write, the more this will make sense. Every time we come across an issue in our code regarding scope, I will discuss it to make things clear. There will be one such issue coming up in the next section. And there are some more C++ staples that blow this issue wide open. They are called **references** and **pointers**, and we will learn about them in *Chapters 9* and *10*, respectively.

A final word on functions – for now

There is even more we could learn about functions, but we know enough about them already to implement the next part of our game. And don't worry if all the technical terms like parameters, signatures, definitions, and so on have not completely sunk in. The concepts will become clearer when we start to use them.

In addition, it has probably not escaped your attention that we have been calling functions, especially the SFML functions, by appending the name of an object and a period before the function name, like this:

```
spriteBee.setPosition...
window.draw...
// etc
```

And yet, our entire discussion of functions saw us calling functions without any objects. What's that all about then? We can write functions as part of a class or simply as a standalone function, as we have seen in this chapter. When we write a function as part of a class, we need an object of that class to call the function, and when we have a standalone function (as we have seen), we don't.

We will write a standalone function in a minute, and we will write classes with functions starting in *Chapter 6*, *Object-Oriented Programming – Starting the Pong Game*. Everything we know so far about functions is relevant in both cases. Only the context changes.

Finally, we can use what we have learned to grow the branches on our tree.

Growing the branches

Next, as I have been promising for around the last 20 pages, we will use all the new C++ techniques – loops, arrays, enumerations, and functions – to draw and move some branches on our tree.

Add this code outside of the `main` function. Just to be absolutely clear, I mean *before* the `int main()` code:

```
#include <sstream>
#include <SFML/Graphics.hpp>
using namespace sf;
// Function declaration
void updateBranches(int seed);
const int NUM_BRANCHES = 6;
Sprite branches[NUM_BRANCHES];
// Where is the player/branch?
// Left or Right
enum class side { LEFT, RIGHT, NONE };
side branchPositions[NUM_BRANCHES];
int main()
{
```

We just achieved quite a few things with that new code:

- First, we wrote a function prototype for a function called `updateBranches`. We can see that it does not return a value (`void`) and it takes an `int` argument called `seed`. We will write the function definition soon and we will then see exactly what it does.
- Next, we declare a constant `int` called `NUM_BRANCHES` and initialize it to `6`. There will be six moving branches on the tree, and we will soon see how `NUM_BRANCHES` is useful to us.
- Following this, we declare an array of `Sprite` objects called `branches` that can hold six `Sprite` instances.

- After that, we declare a new enumeration called `side` with three possible values, `LEFT`, `RIGHT`, and `NONE`. This will be used to describe the position of individual branches, as well as the player, in a few places throughout our code.
- Finally, in the preceding code, we initialize an array of `side` types, with a size of `NUM_BRANCHES` (6). To be clear about what this achieves, we will have an array called `branchPositions` with six values in it. Each of these values is of type `side`, and each can hold the values of either `LEFT`, `RIGHT`, or `NONE`.

Of course, what you really want to know is why the constant, two arrays, and the enumeration were declared *outside* of the `main` function. By declaring them above `main`, they now have **global scope**. Or describing it another way, the constant, two arrays, and the enumeration have scope for the entire game. This will mean we can access and use them all, anywhere in the `main` function and the `updateBranches` function. Note that it is good practice to make all variables as local to where they are used as possible. It might seem useful to make everything global, but this leads to hard-to-read and error-prone code.

Preparing the branches

Now, we will prepare our six `Sprite` objects and load them into the `branches` array. Add the highlighted code just before our game loop:

```
// Position the text
FloatRect textRect = messageText.getLocalBounds();
```

```
  messageText.setOrigin(textRect.left +
    textRect.width / 2.0f,
    textRect.top +
    textRect.height / 2.0f);
  messageText.setPosition(1920 / 2.0f, 1080 / 2.0f);
  scoreText.setPosition(20, 20);
  // Prepare 5 branches
  Texture textureBranch;
  textureBranch.loadFromFile("graphics/branch.png");
  // Set the texture for each branch sprite
  for (int i = 0; i < NUM_BRANCHES; i++) {
    branches[i].setTexture(textureBranch);
    branches[i].setPosition(-2000, -2000);
   // Set the sprite's origin to dead centre
   // We can then spin it round without changing its position
    branches[i].setOrigin(220, 20);
  }
  while (window.isOpen())
  {
```

In the preceding code, first, we declare an SFML `Texture` object and load the `branch.png` graphic into it.

Next, we create a `for` loop that sets `i` to zero and increments `i` by one each pass through the loop, until `i` is no longer less than `NUM_BRANCHES`. This is exactly right because `NUM_BRANCHES` is 6 and the `branches` array has positions 0 through 5.

Inside the `for` loop, we set the `Texture` for each `Sprite` in the `branches` array with `setTexture` and then hide it off-screen with `setPosition`.

Finally, we set the origin (the point that is used to locate the sprite when it is drawn) with `setOrigin` to the center of the sprite. Soon, we will be rotating these sprites, and having the origin in the center means they will spin nicely around, without moving the sprite out of position.

Updating the branch sprites in each frame

In this next code, we set the position of all the sprites in the `branches` array, based on their position in the array and the value of `side` in the corresponding `branchPositions` array. Add the highlighted code and try to understand it, then we can go through it in detail:

```
  // Update the score text
   std::stringstream ss;
   ss << "Score: " << score;
   scoreText.setString(ss.str());
 // update the branch sprites
 for (int i = 0; i < NUM_BRANCHES; i++)
   {
 float height = i * 150;
 if (branchPositions[i] == side::LEFT)
    {
 // Move the sprite to the left side
      branches[i].setPosition(610, height);
 // Flip the sprite round the other way
      branches[i].setRotation(180);
    }
 else if (branchPositions[i] == side::RIGHT)
    {
 // Move the sprite to the right side
      branches[i].setPosition(1330, height);
 // Set the sprite rotation to normal
      branches[i].setRotation(0);
    }
 else
    {
 // Hide the branch
      branches[i].setPosition(3000, height);
    }
  }
} // End if(!paused)
/*
```

```
    ****************************************
    Draw the scene
    ****************************************
```

The code we just added is one big `for` loop that sets `i` to zero and increments `i` by one each time through the loop and keeps going until `i` is no longer less than 6.

Inside the `for` loop, a new `float` variable called `height` is set to `i * 150`. This means that the first branch will have a height of 0, the second of 150, and the sixth of 750.

Next, we have a structure of `if` and `else` blocks. Look at the structure with the code stripped out:

```
if()
{
}
else if()
{
}
else
{
}
```

The first `if` uses the `branchPositions` array to see whether the current branch should be on the left. If it should, it sets the corresponding `Sprite` from the `branches` array to a position on the screen, appropriate for the left (610 pixels) and whatever the current `height` is. It then flips the Sprite by 180 degrees because the `branch.png` graphic "hangs" to the right by default.

The `else if` only executes if the branch is not on the left. This part of the code then uses the same method to see if it is on the right. If it is, then the branch is drawn on the right (1330 pixels). Then, the sprite rotation is set to

zero degrees, just in case it had previously been at 180 degrees. If the x coordinate seems a little bit strange, just remember that we set the origin for the branch sprites to their center.

The final `else` assumes, correctly, that the current `branchPosition` must be `NONE` and hides the branch off-screen at 3000 pixels.

At this point, our branches are in position, ready to be drawn.

Drawing the branches

Here, we use another `for` loop to step through the entire `branches` array from 0 to 5 and draw each branch sprite. Add the following highlighted code:

```
// Draw the clouds
window.draw(spriteCloud1);
window.draw(spriteCloud2);
window.draw(spriteCloud3);
// Draw the branches
for (int i = 0; i < NUM_BRANCHES; i++)
{
  window.draw(branches[i]);
}
// Draw the tree
window.draw(spriteTree);
```

Of course, we still haven't written the function that moves all the branches. Once we have written that function, we will also need to work out when and how to call that function. Let's solve the first problem and write the function.

Moving the branches

We have already added the function prototype above the `main` function. Now, we code the actual definition of the function that will move all the branches down by one position each time it is called. We will code this function in two parts so we can more easily examine what is happening.

Add the first part of the `updateBranches` function *after* the closing curly brace of the `main` function:

```
// Function definition
void updateBranches(int seed)
{
  // Move all the branches down one place
for (int j = NUM_BRANCHES-1; j > 0; j--) {
    branchPositions[j] = branchPositions[j - 1];
  }
}
```

In this first part of the function, we simply move all the branches down one position, one at a time, starting with the sixth branch. This is achieved by making the `for` loop count from 5 through to 0. The code `branchPositions[j] = branchPositions[j - 1];` makes the actual move.

The other thing to note with the previous code is that after we have moved the branch in position 4 to position 5, then the branch in position 3 to position 4, and so on, we will need to add a new branch at position 0, which is the top of the tree.

Now we can spawn a new branch at the top of the tree. Add the highlighted code into the `updateBranches` function and then we will talk about it:

```
// Function definition
void updateBranches(int seed)
{
  // Move all the branches down one place
```

```
for (int j = NUM_BRANCHES-1; j > 0; j--) {
    branchPositions[j] = branchPositions[j - 1];
  }
 // Spawn a new branch at position 0
 // LEFT, RIGHT or NONE
 srand((int)time(0)+seed);
 int r = (rand() % 5);
 switch (r) {
 case 0:
    branchPositions[0] = side::LEFT;
 break;
 case 1:
    branchPositions[0] = side::RIGHT;
 break;
 default:
    branchPositions[0] = side::NONE;
 break;
  }
}
```

In the final part of the `updateBranches` function, we use the integer `seed` variable that gets passed in with the function call. We do this to guarantee that the random number seed is always different, and we will see how this value is arrived at in the next chapter.

Next, we generate a random number between 0 and 4 and store the result in the `int` variable `r`. Now, we `switch` using `r` as the expression.

The `case` statements mean that if `r` is equal to 0, then we add a new branch on the left-hand side at the top of the tree. If `r` is equal to 1, then the branch goes on the right. If `r` is anything else (2, 3, or 4), then `default` ensures that no branch at all will be added at the top. This balance of left, right, and none makes the tree seem realistic (for a fake video game tree) and the game works quite well. You could easily change the code to make the branches more frequent or less so.

Even after all this code for our branches, we still can't glimpse a single one of them in the game. This is because we have more work to do before we can call `updateBranches`.

If you want to see a branch now, you can add some temporary code and call the function five times with a unique seed each time, just before the game loop:

```
updateBranches(1);
updateBranches(2);
updateBranches(3);
updateBranches(4);
updateBranches(5);
while (window.isOpen())
{
```

You can now see the branches in their place. But if the branches are to move, we will need to call `updateBranches` on a regular basis.

Figure 4.1: Branches on a tree

Don’t forget to remove the temporary code before moving on.

Now we can turn our attention to the player as well as call the `updateBranches` function for real.

Summary

Although not quite the longest, this was probably the chapter where we covered the most C++. We looked at the different types of loops we can use, like `for` and `while` loops. We studied arrays for handling large amounts of variables and objects without breaking a sweat. We also learned about enumerations and `switch`. Probably the biggest concept in this chapter was functions, which allow us to organize and abstract our game’s code. We will be looking more deeply at functions in a few more places as the book continues.

Now that we have a fully “working” tree, we can finish the game in the last chapter of this project.

Here are some things that might be on your mind.

Frequently asked questions

Q) How does a `for` loop differ from a while loop in C++?

A) Both `for` and `while` loops in C++ are used for repetition, but a `for` loop is typically used when the number of iterations is known in advance and a `for` loop involves three parts (initialization, condition, and iteration). In

contrast, a `while` loop is more flexible and used when the number of iterations is uncertain.

Q) Can functions in C++ return multiple values?

A) No, a function in C++ can only directly return one value. However, multiple values can be simulated by using parameters passed by references, pointers, or objects. References, pointers, and objects are all discussed in depth in upcoming chapters.

Q) Tell me briefly how the CPU stack relates to function calls and loops in C++.

A) The stack is a region of memory used for managing function calls, local variables, and control flow in C++. Both function calls and loops involve the allocation and deallocation of space on the stack to store information such as local variables, parameters, and the return address. It is not necessary to understand this in order to proceed but knowing it exists helps accept some otherwise inexplicable constructs, specifically function syntax from this chapter.

Q) When should I use an enumeration in C++?

A) Enumerations, often abbreviated to **enums** in C++, are useful when you want to represent a set of named constant values. They improve code readability and help prevent the use of invalid values and operations. Enums are sometimes used for menu options in games, or the example we used in this chapter was days of the week. If you see the value WEDNESDAY, it makes it clear what it represents, whereas the value 3 could be Tuesday or even the number of toes on a cute tree-climbing mammal.

Q)How can I prevent an unwanted infinite loop in C++?

A) To avoid unwanted infinite loops, ensure that the loop condition has a way of becoming `false`. For example, in a for loop, make sure the condition will eventually evaluate to `false`. In a `while` loop, ensure that the loop variable is updated or that there is a break statement when a specific condition is met.

5

Collisions, Sound, and End Conditions: Making the Game Playable

This is the final phase of the first project. By the end of this chapter, you will have your first completed game. Once you have Timber!!! up and running, be sure to read the last section of this chapter, as it will suggest ways to make the game better.

We will cover the following topics in this chapter:

- Preparing the player (and other sprites)
- Drawing the player and other sprites
- Handling the player's input
- Handling death
- Simple sound effects
- Improving the game and code

In this chapter, we will reuse the C++ concepts that we have already learned, but we will see the **SFML** (**Simple and Fast Multimedia Library**) sound features for the first time.

Preparing the player (and other sprites)

Let's add the code for the player's sprite, as well as a few more sprites and textures at the same time. The following code adds a gravestone sprite for when the player gets squashed, an axe sprite to chop with, and a log sprite that can whiz away each time the player chops.

Notice that after the `spritePlayer` object, we also declare a `side` variable, `playerSide`, to keep track of where the player is currently standing. Furthermore, we add some extra variables for the `spriteLog` object, including `logSpeedX`, `logSpeedY`, and `logActive`, to store how fast the log will move and whether it is currently moving. The `spriteAxe` also has two related `float` constant variables to remember where the ideal pixel position is on both the left and the right.

Add this next block of code just before the `while(window.isOpen())` code as we have done so often before. Note that all the code in this next listing is new, not just the highlighted code. I haven't provided any extra context for the following block of code, as the `while(window.isOpen())` should be easy to identify. The highlighted code is the code we have just specifically discussed.

Add the entirety of this code, just before the `while(window.isOpen())` line, and make a mental note of the highlighted lines we have briefly discussed. It will make the rest of the chapter's code easier to understand:

```
// Prepare the player
Texture texturePlayer;
texturePlayer.loadFromFile("graphics/player.png");
Sprite spritePlayer;
spritePlayer.setTexture(texturePlayer);
```

```
spritePlayer.setPosition(580, 720);
// The player starts on the left
side playerSide = side::LEFT;
// Prepare the gravestone
Texture textureRIP;
textureRIP.loadFromFile("graphics/rip.png");
Sprite spriteRIP;
spriteRIP.setTexture(textureRIP);
spriteRIP.setPosition(600, 860);
// Prepare the axe
Texture textureAxe;
textureAxe.loadFromFile("graphics/axe.png");
Sprite spriteAxe;
spriteAxe.setTexture(textureAxe);
spriteAxe.setPosition(700, 830);
// Line the axe up with the tree
const float AXE_POSITION_LEFT = 700;
const float AXE_POSITION_RIGHT = 1075;
// Prepare the flying log
Texture textureLog;
textureLog.loadFromFile("graphics/log.png");
Sprite spriteLog;
spriteLog.setTexture(textureLog);
spriteLog.setPosition(810, 720);
// Some other useful log related variables
bool logActive = false;
float logSpeedX = 1000;
float logSpeedY = -1500;
```

Now, we can draw all our new sprites.

Drawing the player and other sprites

Before we add the code to move the player and use all our new sprites, let's draw them. This is so that, as we add code to update/change/move them, we will be able to see what is happening.

Add the highlighted code to draw the four new sprites:

```
// Draw the tree
window.draw(spriteTree);
// Draw the player
window.draw(spritePlayer);
// Draw the axe
window.draw(spriteAxe);
// Draw the flying log
window.draw(spriteLog);
// Draw the gravestone
window.draw(spriteRIP);
// Draw the bee
window.draw(spriteBee);
```

Run the game, and you will see our new sprites in the scene.

Figure 5.1: New sprites in scene

We are really close to a working game now.

Handling the player's input

Lots of different things depend on the movement of the player. These include:

- When to show the axe
- When to begin animating the log
- When to move all the branches down

Therefore, it makes sense to set up **keyboard handling** for the player chopping. Once this is done, we can put all the features we just mentioned into the same part of the code.

Let's think for a moment about how we detect keyboard presses. In each frame, we test whether a particular keyboard key is currently held down.

If it is, we take action. If the *Escape* key is held down, we quit the game, and if the *Enter* key is held down, we restart the game. So far, this has been sufficient for our needs.

However, there is a problem with this approach when we try and handle the chopping of the tree. The problem has always been there; it just didn't matter until now. Depending on how powerful your PC is, the game loop could be executed thousands of times per second. Each and every pass through the game loop when a key is held down is detected, and the related code will execute it.

So actually, every time you press *Enter* to restart the game, you are most likely restarting it well in excess of a hundred times. This is because even the briefest of presses will last a significant fraction of a second. You can verify this by running the game and holding down the *Enter* key. Note that the time-bar doesn't move. This is because the game is being restarted over and over again, hundreds or even thousands of times a second.

If we don't use a different approach for the player chopping, then just one attempted chop will bring the entire tree down. We need to be a bit more sophisticated. What we will do is allow the player to chop, and then when they do so, disable the code that detects a key press. We will then detect when the player removes their finger from a key and then re-enable the detection of key presses. Here are the steps laid out clearly:

1. Wait for the player to use the *left* or *right* arrow keys to chop a log.
2. When the player chops, disable key press detection.
3. Wait for the player to remove their finger from a key.
4. Re-enable chop detection.
5. Repeat from step 1.

This might sound complicated, but with **SFML's** help, it will be straightforward. Let's implement this now, one step at a time.

Add the highlighted line of code that declares a `bool` variable called `acceptInput`, which will be used to determine when to listen for chops and when to ignore them:

```
float logSpeedX = 1000;
float logSpeedY = -1500;
// Control the player input
bool acceptInput = false;
while (window.isOpen())
{
```

Now that we have our Boolean set up, we can move on to handling setting up a new game.

Handling setting up a new game

Ready for us to handle chops, the highlighted code is added to the `if` block that starts a new game:

```
/*
****************************************
Handle the players input
****************************************
*/
if (Keyboard::isKeyPressed(Keyboard::Escape))
{
    window.close();
}
// Start the game
if (Keyboard::isKeyPressed(Keyboard::Return))
{
    paused = false;
    // Reset the time and the score
    score = 0;
    timeRemaining = 6;
 // Make all the branches disappear
 for (int i = 1; i < NUM_BRANCHES; i++)
    {
        branchPositions[i] = side::NONE;
    }
 // Make sure the gravestone is hidden
    spriteRIP.setPosition(675, 2000);
 // Move the player into position
    spritePlayer.setPosition(580, 720);
    acceptInput = true;
}
/*
****************************************
Update the scene
****************************************
*/
```

In the preceding code, we use a `for` loop to prepare the tree without any branches. This is fair to the player, as if the game started with a branch right above their head, it might be considered unsporting. Players are fine with

hard games, but they hate unfair games. Then, we simply move the gravestone off of the screen and the player into their starting location on the left. The last thing the preceding code does is set `acceptInput` to `true`. We are now ready to receive chopping key presses.

Detecting the player chopping

Now, we can prepare to handle the left and right cursor key presses. Add this simple `if` block, which only executes when `acceptInput` is `true`:

```
// Start the game
if (Keyboard::isKeyPressed(Keyboard::Return))
{
    paused = false;
    // Reset the time and the score
    score = 0;
    timeRemaining = 5;
    // Make all the branches disappear
for (int i = 1; i < NUM_BRANCHES; i++)
    {
        branchPositions[i] = side::NONE;
    }
    // Make sure the gravestone is hidden
    spriteRIP.setPosition(675, 2000);
    // Move the player into position
    spritePlayer.setPosition(675, 660);
    acceptInput = true;
}
// Wrap the player controls to
// Make sure we are accepting input
if (acceptInput)
{
 // More code here next...
}
/*
****************************************
Update the scene
```

```
        ****************************************
        */
```

Now, inside the `if` block that we just coded, add the following highlighted code to handle what happens when the player presses the *right* cursor key on the keyboard:

```
// Wrap the player controls to
// Make sure we are accepting input
if (acceptInput)
{
    // More code here next...
 // First handle pressing the right cursor key
 if (Keyboard::isKeyPressed(Keyboard::Right))
    {
 // Make sure the player is on the right
        playerSide = side::RIGHT;

        score ++;
 // Add to the amount of time remaining
        timeRemaining += (2 / score) + .15;
        spriteAxe.setPosition(AXE_POSITION_RIGHT,
            spriteAxe.getPosition().y);
        spritePlayer.setPosition(1200, 720);
 // Update the branches
 updateBranches(score);

 // Set the log flying to the left
        spriteLog.setPosition(810, 720);
        logSpeedX = -5000;
        logActive = true;
        acceptInput = false;
    }
 // Handle the left cursor key
}
```

Quite a bit is happening in that preceding code, so let's go through it. First, we detect if the player has chopped on the right-hand side of the tree. If they

have, then we set `playerSide` to `side::RIGHT`. We will respond to the value of `playerSide` later in the code.

Then, we add 1 to the score with a `score ++`. The next line of code is slightly mysterious, so here it is again as a reminder:

```
  timeRemaining += (2 / score) + .15;
```

It isn't particularly complicated. See if you can work it out for yourself before reading on. All that is happening is that we add to the amount of time remaining with `timeRemaining +=...`, and therefore, we reward the player for taking action. However, the problem for the player is that the bigger the score gets, the less additional time is added on. By dividing 2 by a score like this (`2 / score`), the chopping rewards quickly diminish. You can play with this formula to make the game easier or harder.

Next, the axe is moved into its right-hand-side position with `spriteAxe.setPosition`, and the player sprite is moved into its right-hand-side position also.

Next, we call `updateBranches` to move all the branches down one place and spawn a new random branch (or space) at the top of the tree.

Then, `spriteLog` is moved into its starting position, camouflaged against the tree, and its `speedX` variable is set to a negative number so that it whizzes off to the left. Also, `logActive` is set to `true` so that the log-moving code that we will write soon animates the log at each frame.

Finally, `acceptInput` is set to `false`. At this point, no more chops can be made by the player. We have solved the problem of the presses being detected too frequently, and we will see how we re-enable the chopping soon.

Now, still inside the `if(acceptInput)` block that we just coded, add the highlighted code to handle what happens when the player presses the *left* cursor key on the keyboard:

```
 // Handle the left cursor key
 if (Keyboard::isKeyPressed(Keyboard::Left))
    {
 // Make sure the player is on the left
        playerSide = side::LEFT;

        score++;
 // Add to the amount of time remaining
        timeRemaining += (2 / score) + .15;
        spriteAxe.setPosition(AXE_POSITION_LEFT,
            spriteAxe.getPosition().y);
        spritePlayer.setPosition(580, 720);
 // update the branches
 updateBranches(score);
 // set the log flying
        spriteLog.setPosition(810, 720);
        logSpeedX = 5000;
        logActive = true;
        acceptInput = false;
    }
}
```

The previous code is just the same as the code that handles the right-hand-side chop, except that the sprites are positioned differently and the `logSpeedX` variable is set to a positive value so that the log whizzes to the right. This is so because the horizontal coordinates increase as sprites are positioned further to the right.

Next, let's see how to detect that a key is released.

Detecting a key being released

To make the preceding code work beyond the first chop, we need to detect when the player releases a key and then set `acceptInput` back to `true`.

This is slightly different from the key handling we have seen so far. **SFML** has two different ways of detecting keyboard input from the player. The first way we have already seen. It is dynamic and instantaneous, exactly what we need to respond immediately to a key press.

The following code uses the other method. Enter the next highlighted code at the top of the `Handle the players input` section, and then we will go through it:

```
/*
****************************************
Handle the players input
****************************************
*/
Event event;
while (window.pollEvent(event))
{
 if (event.type == Event::KeyReleased && !paused)
    {
 // Listen for key presses again
        acceptInput = true;
 // hide the axe
        spriteAxe.setPosition(2000,
            spriteAxe.getPosition().y);
    }
}
if (Keyboard::isKeyPressed(Keyboard::Escape))
{
    window.close();
}
```

First, we declare an object of type `Event` called `event`. Then, we call the `window.pollEvent` function, passing in our new object, `event`. The

`pollEvent` function puts data into the `event` object that describes an operating system event. This could be a key press, key release, mouse movement, mouse click, game controller action, or something that happened to the window itself (it was resized, moved, etc.).

The reason that we wrap our code in a `while` loop is that there might be many events stored in a queue. The `window.pollEvent` function will load them, one at a time, into `event`. Upon each pass through the loop, we will see if we are interested in the current event and respond if we are. When `window.pollEvent` returns `false`, that means there are no more events in the queue and the `while` loop will exit.

Astute readers will notice something is a bit different compared to our discussion of functions. What is happening will be fully explained when we discuss references in *Chapter 9*. As a quick explanation, it is possible to pass a value into a function, and the called function can alter that value so that the new value is available to the calling function. This is done through the concept of references, not by returning a value with a `return` statement.

This `if` condition `(event.type == Event::KeyReleased && !paused)` is `true` when both a key has been released and the game is not paused.

Inside the `if` block, we set `acceptInput` back to `true` and hide the axe sprite off-screen.

You can run the game now and gaze in awe at the moving tree, swinging axe, and animated player. However, it won't squash the player, and the log needs to move when chopped as well.

Animating the chopped logs and the axe

When the player chops, `logActive` is set to `true`, so we can wrap some code in a block that only executes when `logActive` is `true`. Furthermore, each chop sets `logSpeedX` to either a positive or negative number, so the log is ready to start flying away from the tree in the correct direction.

Add the following highlighted code, right after we update the branch sprites:

```
 // update the branch sprites
for (int i = 0; i < NUM_BRANCHES; i++)
    {
float height = i * 150;
        if (branchPositions[i] == side::LEFT)
        {
            // Move the sprite to the left side
            branches[i].setPosition(610, height);
            // Flip the sprite round the other way
            branches[i].setRotation(180);
        }
        else if (branchPositions[i] == side::RIGHT)
        {
            // Move the sprite to the right side
            branches[i].setPosition(1330, height);
            // Flip the sprite round the other way
            branches[i].setRotation(0);
        }
        else
        {
            // Hide the branch
            branches[i].setPosition(3000, height);
        }
    }
 // Handle a flying log
 if (logActive)
    {
        spriteLog.setPosition(
            spriteLog.getPosition().x +
            (logSpeedX * dt.asSeconds()),

        spriteLog.getPosition().y +
            (logSpeedY * dt.asSeconds()));
```

```
        // Has the log reached the right hand edge?
        if (spriteLog.getPosition().x < -100 ||
                    spriteLog.getPosition().x > 2000)
                {
        // Set it up ready to be a whole new log next frame
                    logActive = false;
                    spriteLog.setPosition(810, 720);
                }
            }
        } // End if(!paused)
        /*
        ****************************************
        Draw the scene
        ****************************************
        */
```

The code sets the position of the sprite by getting its current horizontal and vertical location with `getPosition` and then adding to it using `logSpeedX` and `logSpeedY`, respectively, multiplied by `dt.asSeconds`.

After the log sprite has been moved to each frame, the code uses an `if` block to see if the sprite has disappeared out of view on either the left or the right. If it has, the log is moved back to its starting point, ready for the next chop.

If you run the game, you will be able to see the log flying off to the appropriate side of the screen.

Figure 5.2: Flying log

Now for a more sensitive subject. Let's see how we deal with the player losing.

Handling death

Every game must end badly, with either the player running out of time (which we have already handled) or getting squashed by a branch. The mayfly is an aquatic creature that lives anywhere between a few hours and a few days. Playing the Timber!!! game is like being a mayfly in a hurry – you're either running out of time or feeling the branch of destiny squashing your hopes! Our hero in the Timber!!! game may only last a few seconds, and even an experienced player will struggle to last more than a few minutes.

Fortunately, detecting the player getting squashed is really simple. All we want to know is whether the last branch in the `branchPositions` array equals

`playerSide`. If it does, the player is dead.

Add the highlighted code that detects this, and then we will discuss everything we need to do when the player is squashed:

```
 // Handle a flying log
if (logActive)
    {
        spriteLog.setPosition(
            spriteLog.getPosition().x +
                (logSpeedX * dt.asSeconds()),

        spriteLog.getPosition().y +
            (logSpeedY * dt.asSeconds()));
        // Has the log reached the right-hand edge?
if (spriteLog.getPosition().x < -100 ||
            spriteLog.getPosition().x > 2000)
        {
            // Set it up ready to be a whole new cloud next fram
            logActive = false;
            spriteLog.setPosition(800, 600);
        }
    }
 // has the player been squished by a branch?
 if (branchPositions[5] == playerSide)
    {
 // death
        paused = true;
        acceptInput = false;

 // Draw the gravestone
        spriteRIP.setPosition(525, 760);
 // hide the player
        spritePlayer.setPosition(2000, 660);
 // Change the text of the message
        messageText.setString("SQUISHED!!");
 // Center it on the screen
        FloatRect textRect = messageText.getLocalBounds();
        messageText.setOrigin(textRect.left +
            textRect.width / 2.0f,
            textRect.top + textRect.height / 2.0f);
```

```
        messageText.setPosition(1920 / 2.0f,
 1080 / 2.0f);
    }
} // End if(!paused)
/*
****************************************
Draw the scene
****************************************
*/
```

The first thing the new code does, after the player's demise, is set `paused` to `true`. Now, the loop will complete this frame, and it won't run the update part of the loop again until a new game is started by the player.

Then, we move the gravestone into a position near where the player was standing and hide the player's sprite off-screen.

We set the string of `messageText` to `"Squished!!"` and then use the usual technique to center it on the screen.

You can now run the game and play it for real. This image shows the player's final score and their gravestone, as well as the **SQUISHED** message.

Figure 5.3: Squished

There is just one more problem. No, not that the man has left his axe stuck in the tree. Is it just me, or is the game a little bit quiet?

Simple sound effects

We will add three sounds. Each sound will be played on a particular game event: a simple thud sound whenever the player chops, a gloomy losing sound when the player runs out of time, and a retro crushing sound when the player is squashed to death.

How SFML sound works

SFML plays sound effects using two different classes. The first class is the `SoundBuffer` class. This is the class that holds the actual audio data from the sound file. It is `SoundBuffer` that is responsible for loading the `.wav` files

into the PC's **RAM**, in a format that can be played without any further decoding work.

When we write code for the sound effects shortly, we will see that once we have a `SoundBuffer` object with our sound stored in it, we will then create another object of type `Sound`. We can then associate this `Sound` object with a `SoundBuffer` object.

Then, at the appropriate moment in our code, we will be able to call the `play` function of the appropriate `Sound` object.

When to play the sounds

As we will see very soon, the C++ code to load and play sounds is really simple. What we need to consider, however, is *when* we call the `play` function. Where in our code will we put the function calls to `play`?

- The chop sound can be called from the key presses of the left and right cursor keys.
- The death sound can be played from the `if` block that detects that a tree has mangled the player.
- The out-of-time sound can be played from the `if` block that detects that `timeRemaining` is less than zero.

Now, we can write our sound code.

Adding the sound code

First, we add another `#include` directive to make the SFML sound-related classes available. Add the highlighted code:

```
#include <sstream>
#include <SFML/Graphics.hpp>
#include <SFML/Audio.hpp>
using namespace sf;
```

Now, we declare three different `SoundBuffer` objects, load three different sound files into them, and associate three different objects of type `Sound` with the related objects of type `SoundBuffer`. Add the highlighted code:

```
// Control the player input
bool acceptInput = false;
// Prepare the sound
SoundBuffer chopBuffer;
chopBuffer.loadFromFile("sound/chop.wav");
Sound chop;
chop.setBuffer(chopBuffer);
SoundBuffer deathBuffer;
deathBuffer.loadFromFile("sound/death.wav");
Sound death;
death.setBuffer(deathBuffer);
// Out of time
SoundBuffer ootBuffer;
ootBuffer.loadFromFile("sound/out_of_time.wav");
Sound outOfTime;
outOfTime.setBuffer(ootBuffer);
while (window.isOpen())
{
```

Now, we can play our first sound effect. Add the single line of code, as shown next, to the `if` block that detects that the player has pressed the right cursor key:

```
// Wrap the player controls to
// Make sure we are accepting input
if (acceptInput)
{
```

```
    // More code here next...
// First handle pressing the right cursor key
if (Keyboard::isKeyPressed(Keyboard::Right))
    {
        // Make sure the player is on the right
        playerSide = side::RIGHT;

        score++;
        timeRemaining += (2 / score) + .15;
        spriteAxe.setPosition(AXE_POSITION_RIGHT,
            spriteAxe.getPosition().y);
        spritePlayer.setPosition(1120, 660);
        // update the branches
updateBranches(score);

        // set the log flying to the left
        spriteLog.setPosition(800, 600);
        logSpeedX = -5000;
        logActive = true;
        acceptInput = false;
 // Play a chop sound
        chop.play();
    }
```

Add exactly the same code at the end of the next block of code that starts `if (Keyboard::isKeyPressed(Keyboard::Left))` to make a chopping sound when the player chops on the left-hand side of the tree.

Find the code that deals with the player running out of time, and add the highlighted code shown here to play the out-of-time-related sound effect:

```
if (timeRemaining <= 0.f) {
    // Pause the game
    paused = true;
    // Change the message shown to the player
```

```
        messageText.setString("Out of time!!");
        //Reposition the text based on its new size
        FloatRect textRect = messageText.getLocalBounds();
        messageText.setOrigin(textRect.left +
            textRect.width / 2.0f,
            textRect.top +
            textRect.height / 2.0f);
        messageText.setPosition(1920 / 2.0f, 1080 / 2.0f);
     // Play the out of time sound
        outOfTime.play();
    }
```

Finally, to play the death sound when the player is squished, add the highlighted code to the `if` block that executes when the bottom branch is on the same side as the player:

```
    // has the player been squished by a branch?
    if (branchPositions[5] == playerSide)
    {
        // death
        paused = true;
        acceptInput = false;

        // Draw the gravestone
        spriteRIP.setPosition(675, 660);
        // hide the player
        spritePlayer.setPosition(2000, 660);
        messageText.setString("SQUISHED!!");
        FloatRect textRect = messageText.getLocalBounds();
        messageText.setOrigin(textRect.left +
            textRect.width / 2.0f,
            textRect.top + textRect.height / 2.0f);
        messageText.setPosition(1920 / 2.0f, 1080 / 2.0f);
     // Play the death sound
        death.play();
    }
```

If you are having any problems with the sounds not playing, the most likely problem is the sound files not loading. The way to determine if this is happening is to wrap the code that loads the sounds in an `if` block.

First, add the `include` directive that will allow us to use the `cout <<` function, as we did in *Chapter 3* when learning about string concatenation. Here is a reminder of what to add along with the existing `include` directives:

```
#include<iostream>
```

Now, wrap each `loadFromFile` function call, as shown here:

```
if (!chopBuffer.loadFromFile("sound/chop.wav"))
{
    std::cout << "didn't load chop.wav";
}
```

Now, you will get a nice, neat error message telling you if a file didn't load. If it didn't load, check the following:

- The file is named exactly as stated in the code, `chop.wav`.
- The file is in the `sound` folder.
- The `sound` folder is in the project root folder with the C++ file `Timber.cpp`.

That's it! We have finished the first game. Let's discuss some possible enhancements before we move on to the second project.

Improving the game and code

Take a look at these suggested enhancements for the Timber!!! project. You can see the enhancements in action in the `Runnable` folder of the download

bundle:

- **Speed up the code**: There is a part of our code that is slowing down our game. It doesn't matter for this simple game, but we can speed things up by putting the `sstream` code in a block that only executes occasionally. After all, we don't need to update the score thousands of times a second!
- **Debugging console**: Let's add some more text so that we can see the current frame rate. As with the score, we don't need to update this too often. Once every hundred frames will do.
- **Add more trees in the background**: Simply add some more tree sprites and draw them in whatever position looks good (some nearer the camera, and some further away).
- **Improve the visibility of the HUD text**: We can draw simple `RectangleShape` objects behind the score and the FPS counter. Black with a bit of transparency will look quite good.
- **Make the cloud code more efficient**: As we alluded to a few times already, we can use our knowledge of arrays to make the cloud code a lot shorter.

Here is the cloud code using arrays instead of repeating the code three times, once for each cloud:

```
for (int i = 0; i < NUM_CLOUDS; i++)
    {
        clouds[i].setTexture(textureCloud);
        clouds[i].setPosition(-300, i * 150);
        cloudsActive[i] = false;
        cloudSpeeds[i] = 0;

    }

    // 3 New sprites withe the same texture
```

```
//Sprite spriteCloud1;
//Sprite spriteCloud2;
//Sprite spriteCloud3;
//spriteCloud1.setTexture(textureCloud);
//spriteCloud2.setTexture(textureCloud);
//spriteCloud3.setTexture(textureCloud);
// Position the clouds off screen
//spriteCloud1.setPosition(0, 0);
//spriteCloud2.setPosition(0, 150);
//spriteCloud3.setPosition(0, 300);
// Are the clouds currently on screen?
//bool cloud1Active = false;
//bool cloud2Active = false;
//bool cloud3Active = false;
// How fast is each cloud?
//float cloud1Speed = 0.0f;
//float cloud2Speed = 0.0f;
//float cloud3Speed = 0.0f;
```

In the preceding code, the old, unused code is commented out and the new array-based code is at the top. Obviously, you would usually delete the unneeded code. I just left it there to show you. You can view the entire code for the enhanced edition, including the array declaration and initialization, in the *Chapter 5* folder in the file `enhanced.cpp`.

Take a look at the game in action with extra trees, clouds, and a transparent background for the text:

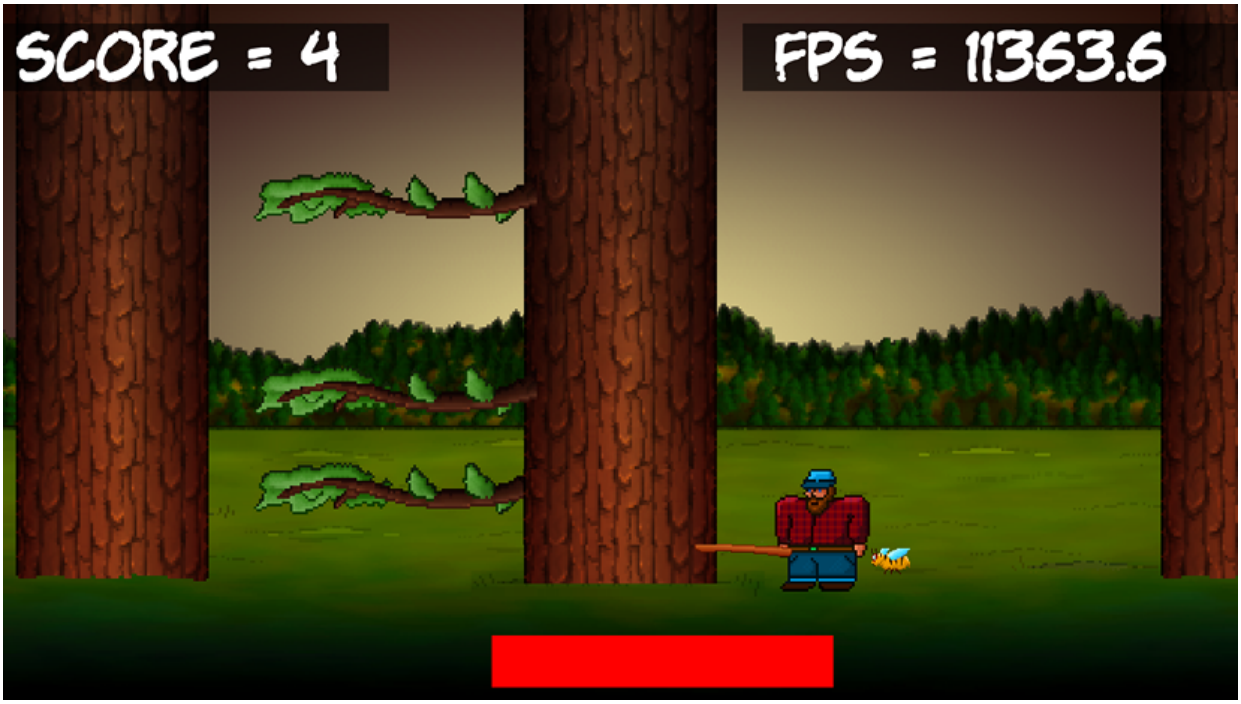


Figure 5.4: Timber enhanced

To see the code for these enhancements, take a look in the `Timber Enhanced Version` folder of the download bundle.

Summary

In this chapter, we added the finishing touches and graphics to the Timber!!! game. If, before this book, you had never coded a single line of C++, then you can give yourself a big pat on the back. In just five modest chapters, you have gone from zero knowledge to a working game.

However, we will not be congratulating ourselves for too long because, in the next chapter, we will move straight on to some slightly more hardcore C++. While the next game, a simple Pong game, is in some ways simpler than Timber!!! was, what we have learned about writing our own classes will prepare us to build more complicated and full-featured games.

Frequently asked questions

Q) The array solution for the clouds was more efficient. But do we really need three separate arrays, one for active, one for speed, and one for the sprite itself?

A) If we look at the properties/variables that various objects have, for example, `sprite` objects, we can see that they are numerous. Sprites have a position, color, size, rotation, and much more. But it would be just perfect if they had `active`, `speed`, and perhaps more as well. The problem is that the coders at SFML can't possibly predict all the ways that we will want to use their `Sprite` class. Fortunately, we can make our own classes. We could make a class called `Cloud` that has a Boolean for `active` and an `int` for speed. We could even give our `Cloud` class an SFML `Sprite` object. We could then simplify our cloud code even further. We will look at designing our own classes in the next chapter.

6

Object-Oriented Programming – Starting the Pong Game

In this chapter, there's a little bit of theory, but the theory will give us the knowledge that we need to start using **Object-Oriented Programming** (**OOP**). OOP helps us organize our code into human-recognizable structures and handle complexity. We will not waste any time in putting that theory to good use as we will use it to code the next project, a Pong game. We will get to look behind the scenes at how we can create new C++ types that we can use as objects. We will achieve this by coding our first class. To get started, we will look at a simplified Pong game scenario so that we can learn about some class basics, and then we will start again and code a Pong game for real using the principles we have learned.

In this chapter, we will cover the following topics:

- Object-object programming: Discuss the staples of encapsulation, polymorphism, and inheritance, and why we would want to use OOP at all
- The theory of a Pong bat: Learn about OOP and classes using a hypothetical `Bat` class
- Creating the Pong project

- Coding the `Bat` class: Start working on the Pong game including coding a real `Bat` class to represent the player's bat
- Using the `Bat` class and coding the main function

The following are our four projects for this book: https://github.com/PacktPublishing/Beginning-C-Game-Programming-Third-Edition/tree/main/Pong

Object-oriented programming

Object-oriented programming is a programming paradigm that we could consider to be almost the standard way to code. It is true there are non-OOP ways to code and there are even some non-OOP game coding languages/libraries. However, since we are starting from scratch, there is no reason to do things in any other way.

OOP will do the following:

- Make our code easier to manage, change, or update
- Make our code quicker and more reliable to write
- Make it possible to easily use other people's code (like we have with SFML)

We have already seen the third benefit in action. Now, let's discuss exactly what OOP is.

OOP is a way of programming that involves breaking our requirements down into chunks that are more manageable than the whole. Each chunk is self-contained, yet it works with the other parts of our program. Furthermore, it can also be used by other programs. These chunks are what we have been referring to as objects.

When we plan and code an object, we do so with a **class**.

A **class** can be thought of as the blueprint of an object.

We implement an object of a class. This is called an **instance** of a class. Think about a house blueprint. You can't live in it, but you can build a house from it. You build an instance of the house. Often, when we design classes for our games, we write them to represent real-world *things*. In the next project, we will write classes for a bat that the player controls and a ball that the player can bounce around the screen with the bat. However, OOP is more than this.

OOP is a *way* of doing things, a methodology that defines best practices.

The three core principles of OOP are **encapsulation**, **polymorphism**, and **inheritance**. This might sound complex but, taken a step at a time, this is reasonably straightforward.

Encapsulation

Encapsulation means keeping the internal workings of your code safe from interference from the code that uses it. You can achieve this by allowing only the variables and functions you choose to be accessed. This means your code can always be updated, extended, or improved without affecting the programs that use it, provided the exposed parts are still accessed in the same way. C++ achieved encapsulation with the use of the public and private keywords. We will see these in action soon.

As an example, with proper encapsulation, it wouldn't matter if the SFML team needed to update the way their `Sprite` class works. If the function signatures remain the same, they don't have to worry about what goes on inside. The code that we wrote before the update will still work after the update.

OOP doesn't eliminate the need for careful planning before we write our code; instead, encapsulation provides a way to structure code that can potentially make our planning more successful. This is even more the case if we are working as a team.

Polymorphism

Polymorphism allows us to write code that is less dependent on the *types* we are trying to manipulate. This will make our code clearer and more efficient. Polymorphism means *different forms*. If the objects that we code can be more than one type of thing, then we can take advantage of this. Polymorphism might sound a little bit like black magic at this point. The classic example of polymorphism is the relationship between different animals in the animal kingdom. What if we are making a zoo game and we make a whole bunch of arrays, functions, and variables for the elephants? Quite quickly, it becomes apparent that we now need to write arrays, functions, and variables for lions, then tigers, etc. What if we could write one set of arrays, functions, and variables that worked with all animals? With polymorphism, we can write for a generic animal object and use it with all our zoo-based classes. We will use polymorphism in the final project, and everything will become clearer.

Inheritance

Just like it sounds, **inheritance** means we can harness all the features and benefits of other people's classes, including encapsulation and polymorphism, while further refining their code specifically to our situation. If we were writing a countryside simulator, we could probably make use of the animal-based code from the zoo game. We will use inheritance for the first time at the same time as we use polymorphism.

Why use OOP?

When written properly, OOP allows you to add new features without worrying about how they interact with existing features. When you do have to change a class, its self-contained (encapsulated) nature means less or perhaps even zero consequences for other parts of the program.

You can use other people's code (like the SFML classes) without knowing or perhaps even caring about how it works inside.

OOP (and, by extension, SFML) allows you to write games that use complicated concepts such as multiple cameras, multiplayer, OpenGL, directional sound, and more besides—all of this without breaking a sweat.

You can create multiple similar yet different versions of a class without starting the class from scratch by using inheritance.

You can still use the functions intended for the original type of object with your new object because of polymorphism.

All this makes sense really and means that we have loads more time to concentrate on the unique aspects of our own programs. And as we know, C++ was designed from the start with all this OOP in mind.

The ultimate key to success with OOP and making games (or any other type of app), other than the determination to succeed, is planning and design. It is not so much just "knowing" all the C++, SFML, and OOP topics that will help you to write great code but, rather, applying all that knowledge to write code that is well structured/designed. The code in this book is presented in an order and manner that's appropriate for learning about the various C++ topics in a gaming context. The art and science of structuring your code is called **design patterns**. As your code gets longer and more complex, effective use of design patterns will become more important. The good news is that we don't need to invent these design patterns ourselves. We will need to learn about them as our projects get more complex. As our projects become more complex, our design patterns will evolve too.

In this project, we will learn about and use basic classes and encapsulation. As this book progresses, we will get a bit more daring and use inheritance, polymorphism, and other OOP-related C++ features too.

What exactly is a class?

A **class** is a bunch of code that can contain functions, variables, loops, and all the other C++ syntax we have already learned about. Each new class will be declared in its own `.h` code file with the same name as the class, while its functions will be defined in their own `.cpp` file. The syntax we will

apply to the definitions in the `.cpp` file will make it clear that they are part of the class declared in the `.h` file.

When we use a function in a class, it is a specialized type of function often referred to as a **method**. For simplicity, I will continue to refer to all functions as *functions*, but you could call the functions of our classes *methods* if you wish.

Once we have written a class, we can use it to make as many objects from it as we want. Remember, the class is the blueprint, and we make objects based on the blueprint. The house isn't the blueprint, just like the object isn't the class. It is an object made *from* the class.

You can think of an object as a **variable** and the class as a **type**.

Of course, with all this talk of OOP and classes, we haven't actually seen any code. Let's fix that now.

The theory of a Pong bat

What follows is a hypothetical discussion of how we might use OOP to get started with the Pong project by coding a `Bat` class. Don't add any code to the project just yet as what follows is over-simplified to explain the theory. Later in this chapter, we will code it for real. When we get to coding the class for real, it will be different, but the principles we will learn about here will prepare us for success.

We will begin by exploring variables and functions (or methods) as part of a class.

Declaring the class, variables, and functions

A bat is a real-world thing that has properties, behavior, and a specific appearance. It performs a role; it bounces a ball when it collides with the ball. A bat that bounces a ball is, therefore, an excellent first candidate for a class.

If you don't know what Pong is, then take a look at this link: https://en.wikipedia.org/wiki/Pong.

Let's take a look at a hypothetical `Bat.h` file:

```
class Bat
{
    private:
        // Length of the pong bat
int m_Length = 100;
        // Height of the pong bat
int m_Height = 10;
        // Location on x axis
int m_XPosition;
        // Location on y axis
int m_YPosition;
    public:
        void moveRight();
        void moveLeft();
};
```

At first glance, the code might appear a little complex, but when it has been explained, we will see there are very few concepts we haven't already covered.

The first thing to notice is that a new class is declared using the `class` keyword followed by the name of the class and that the entire declaration is enclosed in curly braces followed by a closing semicolon:

```
class Bat
{
    …
    …
};
```

Now, let's look at the variable declarations and their names:

```
// Length of the pong bat
int m_Length = 100;
// Height of the pong bat
int m_Height = 10;
// Location on x axis
int m_XPosition;
// Location on y axis
int m_YPosition;
```

All the names are prefixed with `m_`. This `m_` prefix is not compulsory, but it is a good convention. Although this naming convention is not enforced by the C++ language itself, it is widely adopted in the C++ community for class data members. Variables that are declared as part of the class are called **member variables**. Prefixing with an `m_` makes it plain when we are dealing with a member variable. When we write functions for our classes, we will start to see **local** (non-member) variables and **parameters** as well. Then, the `m_` convention will prove itself useful. By adhering to this

convention, you make it immediately apparent that these variables are part of the class, distinguishing them from local variables or parameters. Different projects, companies, and systems use different conventions for variable naming but using some kind of prefix for member variables is an industry best practice.

For example, imagine if you had non-member variables in the same scope without the `m_` prefix, like this:

```
int Length = 50;  // Non-member variable
```

Now, without the `m_` prefix, it becomes less clear whether `Length` is a class member or not. Consistent use of the `m_` prefix helps avoid such confusion, contributing to more maintainable and self-explanatory code.

Also, notice that all the variables are in a section of the code headed with the `private` keyword. Scan your eyes over the previous code and note that the body of the class code is separated into two sections:

```
private:
    // Anything the instances
// cannot directly interact with
public:
    // Variables and functions here can be
// accessed by a user of the instance
```

The `public` and `private` keywords control the encapsulation of our class. Anything that is private cannot be accessed directly by the user of an instance/object of the class. If you are designing a class for others to use, you don’t want them to be able to alter anything at will. Note that member

variables do not have to be private, but good encapsulation is achieved by making them private whenever possible.

This means that our four member variables (`m_Length`, `m_Height`, `m_XPosition`, and `m_YPosition`) cannot be accessed directly by our game engine from the `main` function. They can only be accessed indirectly by the code of the class. This is encapsulation in action. For the `m_Length` and `m_Height` variables, this is easy to accept as long as we don't need to change the size of the bat. The `m_XPosition` and `m_YPosition` member variables, however, need to be accessed, or how on earth will we move the bat?

This problem is solved with the `public` section of the code, as follows:

```
void moveRight();
void moveLeft();
```

The class provides two functions that are public and will be usable with an object of the `Bat` type. When we look at the definitions of these functions, we will see how exactly these functions manipulate the private variables.

In summary, we have a bunch of inaccessible (private) variables that cannot be used from the `main` function. This is good because encapsulation makes our code less error-prone and more maintainable. We then solve the problem of moving the bat by providing indirect access to the `m_XPosition` and `m_YPosition` variables by providing two public functions.

The code in the `main` function can call the public functions using an instance of the class, but the code inside the functions controls exactly how the variables are used.

We can visualize this class information as shown in this next image:

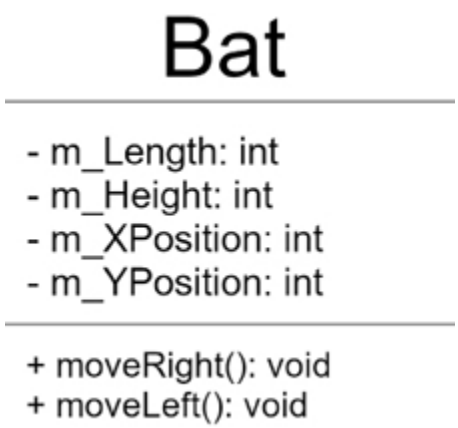


Figure 6.1: Information for the Bat class

In the preceding image, the top section represents the class name `Bat`, and the middle section contains the class's member variables, each preceded by a dash (-) to indicate that they are private. The bottom section contains the class's member functions, each preceded by a plus sign (+) to indicate that they are public. This convention helps quickly convey the access levels of class members, providing a visual representation of encapsulation in the class.

This format of representing a class is part of the **Unified Modeling Language** or **UML**. UML is a huge topic on its own and beyond the scope of this book but understanding that these conventions exist for representing the design decisions in our C++ code is a good start. You can find out more about UML at the official website: https://www.uml.org/.

Let's take a look at the function definitions.

The class function definitions

The function definitions we will write in this book will all go in a separate file to the class and function declarations. We will use files with the same name as the class and the `.cpp` file extension. Remember, this is to keep our

code organized as well as to separate the declarations from the definitions, which can be useful if you want to see what a class does at a glance (the declarations in the `.h` file) rather than studying the details (the definitions in the `.cpp` file). So, for example, the following code would go into a file called `Bat.cpp`. Look at the following code, which has just one new concept:

```
#include "Bat.h"
void Bat::moveRight()
{
    // Move the bat a pixel to the right
    m_XPosition ++;
}
void Bat::moveLeft()
{
    // Move the bat a pixel to the left
    m_XPosition --;
}
```

The first thing to note is that we must use an `include` directive to include the class and function declarations from the `Bat.h` file. This makes the code in the `.cpp` file aware of the declarations in the `.h` file.

The new concept we can see here is the use of the **scope resolution operator**, `::`. Since the functions belong to a class, we must write the signature part slightly different to a standard non-member function by prefixing the function name with the class name, as well as `::`, for example, `void Bat::moveLeft()` and `void Bat::moveRight`.

In this example, `Bat::` before each function name indicates that `moveRight` and `moveLeft` are member functions of the `Bat` class. It explicitly ties these functions to the class declaration, ensuring that the compiler associates them correctly during compilation.

This usage of the scope resolution operator also enhances code clarity and avoids naming conflicts, especially when dealing with multiple classes or functions with similar names.

Actually, we have briefly seen the scope resolution operator before (that is, whenever we declare an object of a class) and we have not previously used `using namespace..`.

Note that we could have put the function definitions and declarations in one file, like this:

```
class Bat
{
    private:
        // Length of the pong bat
int m_Length = 100;
        // Height of the pong bat
int m_Height = 10;
        // Location on x axis
int m_XPosition;
        // Location on y axis
int m_YPosition;
    public:
        void Bat::moveRight()
 {
            // Move the bat a pixel to the right
            m_XPosition ++;
        }
        void Bat::moveLeft()
 {
            // Move the bat a pixel to the left
            m_XPosition --;
        }
};
```

However, when our classes get longer (as they will with our first Zombie Arena game), it is more organized to separate the function definitions into their own file. Furthermore, header files are considered public and are often used for documentation purposes if other people will be using the code that we write.

But how do we use a class once we have coded it?

Using an instance of a class

Despite all the code we have seen related to classes, we haven't actually used a class. We already know how to do this as we have used the SFML classes many times already.

First, we would create an instance of the `Bat` class, like this:

```
Bat bat;
```

The `bat` object has all the variables we declared in `Bat.h`. We just can't access them directly. We can, however, move our bat using its public functions, like this:

```
bat.moveLeft();
```

Or we can move it like this:

```
bat.moveRight();
```

Remember that `bat` is a `Bat` and, as such, it has all the member variables and all of the functions available to it.

Later, we may decide to make our Pong game multiplayer. In the `main` function, we could change the code so that the game has two bats, perhaps like this:

```
Bat bat;
Bat bat2;
```

It is vitally important to realize that each of these instances of `Bat` is a separate object with its very own set of variables, just as our player sprite, tree, bee, and axe sprites were all individual instances of the SFML `Sprite` class. There are more ways to initialize an instance of a class, and we will see an example of this when we code the `Bat` class for real next.

Creating the Pong project

Since setting up a project is a fiddly process, we will go through it step by step, like we did for the Timber!!! project. I won't show you the same screenshots that I did for the Timber!!! project, but the process is the same, so flip back to *Chapter 1*, *Welcome to Beginning C++ Game Programming Third Edition!*, if you want a reminder of the locations of the various project properties:

1. Start Visual Studio and click on the **Create New Project** button. Or, if you still have the Timber!!! project open, you can select **File** | **New project**.
2. In the window that appears, choose **Console app** and click the **Next** button. You will then see the **Configure your new project** window.
3. In the **Configure your new project** window, type `Pong` in the **Project name** field. Note that this causes Visual Studio to automatically

configure the **Solution name** field so that it has the same name.

4. In the **Location** field, browse to the `VS Projects` folder that we created in *Chapter 1*. This will be the location where all our project files will be kept.
5. Check the option **Place solution and project in the same directory**.
6. When you have completed these steps, click **Create**. The project is generated by Visual Studio, including some C++ code in the `main.cpp` file.
7. We will now configure the project to use the SFML files that we put in the `SFML` folder. From the main menu, select **Project** | **Pong properties…**. At this stage, you should have the **Pong Property Pages** window open.
8. In the **Pong Property Pages** window, select **All Configurations** from the **Configuration** dropdown and make sure the **Platform** dropdown is set to **Win32**.
9. Now, select **C/C++** and then **General** from the left-hand menu.
10. After this, locate the **Additional Include Directories** edit box and type the drive letter where your `SFML` folder is located, followed by `\SFML\include`. The full path to type, if you located your `SFML` folder on your `D:` drive, is `D:\SFML\include`. Change your path if you installed SFML on a different drive.
11. Click **Apply** to save your configurations so far.
12. Now, still in the same window, perform these steps. From the left-hand menu, select **Linker** and then **General**.
13. Now, find the **Additional Library Directories** edit box and type the drive letter where your `SFML` folder is, followed by `\SFML\lib`. So, the full path to type if you located your `SFML` folder on your `D:` drive is

`D:\SFML\lib`. Change your path if your SFML files are on a different drive.

14. Click **Apply** to save your configurations so far.
15. Next, still in the same window, perform these steps. Switch the **Configuration** dropdown to **Debug** as we will be running and testing Pong in debug mode.
16. Select **Linker** and then **Input**.
17. Find the **Additional Dependencies** edit box and click on it on the far left-hand side. Now, copy and paste/type in the following: `sfml-graphics-d.lib;sfml-window-d.lib;sfml-system-d.lib;sfml-network-d.lib;sfml-audio-d.lib;`. Be extra careful to place the cursor exactly at the start of the edit box's current content so that you don't overwrite any of the text that is already there.
18. Click **OK**.
19. Click **Apply** and then **OK**.
20. On the main Visual Studio window, next to the **Debug** dropdown, make sure that **x86** is selected, not **x64**.
21. Now, we need to copy the SFML `.dll` files into the main project directory. My main project directory is `D:\VS Projects\Pong`. It was created by Visual Studio in the previous steps. If you put your `VS Projects` folder somewhere else, then perform this step there instead. The files we need to copy into the project folder are located in our `SFML\bin` folder. Open a window for each of the two locations and highlight all the files in the `SFML\bin` folder.
22. Now, copy and paste the highlighted files into the project folder, that is, `D:\VS Projects\Pong`.

We now have the project properties configured and ready to go.

We will be displaying some text for a **Heads Up Display** (**HUD**) in this game that will show the player's score and remaining lives. For this, we need a font.

Download this free-for-personal-use font from http://www.dafont.com/theme.php?cat=302 and unzip the download. Or feel free to use a font of your choice. You will just need to make some minor changes to the code when we load the font.

Create a new folder called `fonts` in the `VS Projects\Pong` folder and add the `DS-DIGIT.ttf` file to the `VS Projects\Pong\fonts` folder.

We are now ready to code our first C++ class.

Coding the Bat class

The simple Pong bat example was a good way of introducing the basics of classes. Classes can be simple and short, like the preceding `Bat` class, but they can also be longer and more complicated and contain other objects made from other classes. Furthermore, there are additional new concepts regarding classes that we will learn about. We will also see and code a **constructor** function that sets up our instances ready for use.

When it comes to making games, there are a few vital things missing from the hypothetical `Bat` class. It might be fine for all these private member variables and public functions, but how will we draw anything? Our Pong bat needs a sprite, and in some games, our classes will also need a texture. Furthermore, we need a way to control the rate of animation of all our game

objects, just like we did with the bee and the clouds in the previous project. We can include other objects in our class in the same way that we included them in the `main.cpp` file. Let's code our `Bat` class for real so that we can see how we can solve all these issues.

Coding Bat.h

To get started, we will code the header file. Right-click on **Header Files** in the **Solution Explorer** window and select **ADD** | **New Item**. Next, choose the **Header File (.h)** option and name the new file `Bat.h`. Click the **Add** button. We are now ready to code the file.

Add the following code to `Bat.h`:

```
#pragma once
#include <SFML/Graphics.hpp>
using namespace sf;
class Bat
{
private:
    Vector2f m_Position;
    // A RectangleShape object
    RectangleShape m_Shape;
    float m_Speed = 1000.0f;
    bool m_MovingRight = false;
    bool m_MovingLeft = false;
public:
    Bat(float startX, float startY);
    FloatRect getPosition();
    RectangleShape getShape();
    void moveLeft();
    void moveRight();
    void stopLeft();
    void stopRight();
    void update(Time dt);
};
```

First, note the `#pragma once` declaration at the top of the file. This prevents the file from being processed by the compiler more than once. As our games get more complicated with perhaps dozens of classes, this will speed up compilation time.

Note the names of the member variables and the parameters and return types of the functions. We have a `Vector2f` called `m_Position`, which will hold the horizontal and vertical position of the player's bat. We also have an SFML `RectangleShape`, which will visually represent the bat that appears on the screen. The `RectangleShape` and `Sprite` classes are both part of the SFML **graphics module** and are used for rendering objects on the screen. `RectangleShape` is primarily used for rendering simple rectangles or squares, whereas `Sprite` is used for rendering textured images. As a Pong bat is a simple white rectangle, I have opted for the `RectangleShape` option.

There are two Boolean members that will track which direction, if any, the bat is currently moving in, and we have a `float` called `m_Speed` that tells us the number of pixels per second at which the bat can move when the player decides to move it left or right.

The next part of the code needs some explanation since we have a function called `Bat`; this is the exact same name as the class. This is called a constructor.

Constructor functions

As a refresher, when a class is coded, a special function is created by the compiler. We don't see this function in our code, but it is there. It is called a constructor. The constructor is provided behind the scenes by the compiler.

It is the function that would have been called if we used our hypothetical `Bat` class example.

When we need to write some code to prepare an object for use, a good place to do this is often in the constructor. When we want the constructor to do anything other than simply create an instance, we must replace the default (unseen) constructor provided by the compiler. This is what we will do with the `Bat` constructor function.

Notice that the `Bat` constructor takes two `float` parameters. Here is the declaration again for convenience:

```
Bat(float startX, float startY);
```

This is perfect for initializing the position on the screen when we first create a `Bat` object. Also note that constructors have no return type, not even `void`.

We will soon use the constructor function, `Bat`, and an initializer list to put this game object into its starting position. Remember that this function is called at the time that an object of the `Bat` type is declared.

Continuing with the Bat.h explanation

Next is the `getPosition` function, which returns a `FloatRect`, representing the four points that define a rectangle. Then, we have `getShape`, which returns a `RectangleShape`. This will be used so that we can return to the main game loop, `m_Shape`, so that it can be drawn.

We also have the `moveLeft`, `moveRight`, `stopLeft`, and `stopRight` functions, which are for controlling if, when, and in which direction the bat

will be in motion.

Finally, we have the `update` function, which takes a `Time` parameter. This function will be used to calculate how to move the bat in each frame. As a bat and a ball will move differently from each other, it makes sense to encapsulate the movement code inside the class. We will call the `update` function once at each frame of the game from the `main` function.

You can probably guess that the `Ball` class will also have an `update` function.

Now, we can code `Bat.cpp`, which will implement all the definitions and use the member variables.

Coding Bat.cpp

Let's create the file, and then we can start discussing the code. Right-click the **Source Files** folder in the **Solution Explorer** window. Now, select **C++ File (.cpp)** and enter `Bat.cpp` in the **Name:** field. Click the **Add** button and our new file will be created for us.

We will divide the code for this file into two parts to make discussing it simpler.

First, code the `Bat` constructor function, as follows:

```
#include "Bat.h"
// This is the constructor and it is called
// when we create an object
Bat::Bat(float startX, float startY) : m_Position(startX, start
{
    m_Shape.setSize(sf::Vector2f(50, 5));
```

```
    m_Shape.setPosition(m_Position);
}
```

In the preceding code, we can see that we include the `bat.h` file. This makes all the functions and variables that were declared previously in `bat.h` available to us.

We implement the constructor because we need to do some work to get the instance set up, and the default unseen empty constructor provided by the compiler is not sufficient. Remember that the constructor is the code that runs when we initialize an instance of `Bat`.

Notice that we use the `Bat::Bat` syntax as the function name to make it clear we are using the `Bat` function from the `Bat` class.

This constructor receives two `float` values, `startX` and `startY`. Next, we see something we haven't seen before. Immediately after the function parameters, we see this code:

```
: m_Position(startX, startY)
```

This is called an **initializer list**. Using member initializer lists is often considered more efficient than initializing variables in the body of the constructor and can be beneficial for certain types of variables. What is happening is we are using a shortened and clearer syntax to initialize the `Vector2f mPosition` with the values passed into the function as parameters.

The `Vector2f` named `m_Position` now holds the values that were passed in, and because `m_Position` is a member variable, these values are accessible throughout the class. Note, however, that `m_Position` was declared as

`private` and will not be accessible in our `main` function file—not directly, anyway. We will see how we can resolve this issue soon.

Finally, in the body of the constructor, we initialize the `RectangleShape` called `m_Shape` by setting its size and position. This is different from how we coded the hypothetical `Bat` class in the *The theory of a Pong bat* section. The SFML `Sprite` class has convenient variables for size and position that we can access using the `setSize` and `setPosition` functions, so we don't need the hypothetical `m_Length` and `m_Height` anymore.

Furthermore, note that we will need to vary how we initialize the `Bat` class (compared to the hypothetical `Bat` class) to suit our custom constructor, and we will see this code soon.

We need to implement the remaining five functions of the `Bat` class. Add the following code to `Bat.cpp` after the constructor we just discussed:

```
FloatRect Bat::getPosition()
{
    return m_Shape.getGlobalBounds();
}
RectangleShape Bat::getShape()
{
    return m_Shape;
}
void Bat::moveLeft()
{
     m_MovingLeft = true;
}
void Bat::moveRight()
{
    m_MovingRight = true;
}
void Bat::stopLeft()
{
    m_MovingLeft = false;
```

```
}
void Bat::stopRight()
{
    m_MovingRight = false;
}
void Bat::update(Time dt)
{
    if (m_MovingLeft) {
        m_Position.x -= m_Speed * dt.asSeconds();
    }
    if (m_MovingRight) {
        m_Position.x += m_Speed * dt.asSeconds();
    }
    m_Shape.setPosition(m_Position);
}
```

Let's go through the code we have just added.

First, we have the `getPosition` function. All this does is return a `FloatRect` to the code that called it. The `m_Shape.getGlobalBounds` line of code returns a `FloatRect` that is initialized with the coordinates of the four corners of the `RectangleShape`, that is, `m_Shape`. We will call this function from the `main` function when we are determining whether the ball has hit the bat.

Next, we have the `getShape` function. All this function does is pass a copy of `m_Shape` to the calling code. This is necessary so that we can draw the bat in the `main` function. When we code a public function with the sole purpose of passing back private data from a class, we call it a getter function.

Now, we can look at the `moveLeft`, `moveRight`, `stopLeft`, and `stopRight` functions. What they do is set the `m_MovingLeft` and `m_MovingRight` Boolean variables appropriately so that they keep track of the player's current movement intentions. Note, however, that they don't do anything to

the `RectangleShape` instance or the `FloatRect` instance, which determine the position. This is just what we need.

The last function in the `Bat` class is `update`. We will call this function once per frame of the game. The `update` function will grow in complexity as our game projects get more complicated. For now, all we need to do is tweak `m_Position`, depending on whether the player is moving left or right. Note that the formula that's used to do this tweak is the same one that we used for updating the bee and the clouds in the Timber!!! project. The code multiplies the speed by the delta time and then adds or subtracts it from the position. This causes the bat to move relative to how long the frame took to update. Next, the code sets the position of `m_Shape` with whatever the latest values held in `m_Position` happen to be.

Having an `update` function in our `Bat` class rather than the `main` function is encapsulation. Rather than updating the positions of all the game objects in the `main` function as we did in the Timber!!! project, each object will be responsible for updating itself.

Using the Bat class and coding the main function

Switch to the `main.cpp` file that was automatically generated when we created the project. If you had a file called `Pong.cpp` automatically created, you can leave it as it is or right-click it in the **Solution Explorer** to rename it `main.cpp`. The only thing that matters is that it has the `main` function in it so that is where execution will begin. Delete all its auto-generated code and add the code that follows.

Code the `Pong.cpp` file as follows:

```
#include "Bat.h"
#include <sstream>
#include <cstdlib>
#include <SFML/Graphics.hpp>
int main()
{
    // Create a video mode object
VideoMode vm(1920, 1080);
    // Create and open a window for the game
RenderWindow window(vm, "Pong", Style::Fullscreen);
    int score = 0;
    int lives = 3;

    // Create a bat at the bottom center of the screen
Bat bat(1920 / 2, 1080 - 20);
    // We will add a ball in the next chapter
// Create a Text object called HUD
    Text hud;
    // A cool retro-style font
    Font font;
    font.loadFromFile("fonts/DS-DIGIT.ttf");
    // Set the font to our retro-style
    hud.setFont(font);
    // Make it nice and big
    hud.setCharacterSize(75);
    // Choose a color
    hud.setFillColor(Color::White);
    hud.setPosition(20, 20);
    // Here is our clock for timing everything
    Clock clock;
    while (window.isOpen())
    {
        /*
        Handle the player input
        ******************************
        ******************************
        ******************************
        */
/*
        Update the bat, the ball and the HUD
        ******************************
        ******************************
```

```
            ****************************
            */
    /*
            Draw the bat, the ball and the HUD
            ****************************
            ****************************
            ****************************
            */

        }
        return 0;
    }
```

In the preceding code, the structure of the main game `while` loop is similar to the one we used in the Timber!!! project. The first exception, however, is when we create an instance of the `Bat` class:

```
// Create a bat
Bat bat(1920 / 2, 1080 - 20);
```

The preceding code calls the constructor function to create a new instance of the `Bat` class. The code passes in the required arguments and allows the `Bat` class to initialize its position in the center of the screen near the bottom. This is the perfect position for our bat to start.

Also note that I have used comments to indicate where the rest of the code will eventually be placed. It is all within the game loop, just like it was in the Timber!!! project. Here is where the rest of the code will go again, just to remind you:

```
 /*
        Handle the player input
        …
        /*
```

```
        Update the bat, the ball and the HUD
        …


        /*
        Draw the bat, the ball and the HUD
        …
```

Next, add the code to the `Handle the player input` section, as follows:

```
Event event;
while (window.pollEvent(event))
{
    if (event.type == Event::Closed)
        // Quit the game when the window is closed
        window.close();
}
// Handle the player quitting
if (Keyboard::isKeyPressed(Keyboard::Escape))
{
    window.close();
}
// Handle the pressing and releasing of the arrow keys
if (Keyboard::isKeyPressed(Keyboard::Left))
{
    bat.moveLeft();
}
else
{
    bat.stopLeft();
}
if (Keyboard::isKeyPressed(Keyboard::Right))
{
    bat.moveRight();
}
else
{
    bat.stopRight();
}
```

The preceding code handles the player quitting the game by pressing the *Escape* key, exactly like it did in the Timber!!! project. Next, there are two `if-else` structures that handle the player moving the bat. Let's analyze the first of these two structures:

```
if (Keyboard::isKeyPressed(Keyboard::Left))
{
    bat.moveLeft();
}
else
{
    bat.stopLeft();
}
```

The preceding code will detect whether the player is holding down the left arrow cursor key on the keyboard. If they are, then the `moveLeft` function is called on the `Bat` instance. When this function is called, the `true` value is set to the `m_MovingLeft` private Boolean variable. If, however, the left arrow key is not being held down, then the `stopLeft` function is called and the `m_MovingLeft` is set to `false`.

The exact same process is then repeated in the next `if-else` block of code to handle the player pressing (or not pressing) the *right arrow* key.

Next, add the following code to the `Update the bat, the ball and the HUD` section, as follows:

```
// Update the delta time
Time dt = clock.restart();
bat.update(dt);
// Update the HUD text
std::stringstream ss;
```

```
    ss << "Score:" << score << "  Lives:" << lives;
    hud.setString(ss.str());
```

In the preceding code, we use the exact same timing technique that we used for the Timber!!! project, only this time, we call `update` on the `Bat` instance and pass in the delta time. Remember, when the `Bat` class receives the delta time, it will use the value to move the bat based on the previously received movement instructions from the player and the desired speed of the bat.

Next, add the following code to the `Draw the bat, the ball and the HUD` section, as follows:

```
    window.clear();
    window.draw(hud);
    window.draw(bat.getShape());
    window.display();
```

In the preceding code, we clear the screen, draw the text for the HUD, and use the `bat.getShape` function to grab the `RectangleShape` instance from the `Bat` instance and draw it to the screen. Finally, we call `window.display`, just like we did in the previous project, to draw the bat in its current position.

At this stage, you can run the game and you will see the HUD and a bat. The bat can be moved smoothly left and right using the arrow/cursor keys:

Figure 6.2: Our Pong game, complete with HUD and bat

Congratulations! That is the first class all coded and deployed.

Summary

In this chapter, we discovered the basics of OOP, such as how to code and use a class, including making use of encapsulation to control how code outside of our classes can access the member variables but only to the extent and in the manner that we want it to. This is just like SFML classes, which allow us to create and use `Sprite` and `Text` instances, but only in the way they were designed to be used.

Don't concern yourself too much if some of the details around OOP and classes are not entirely clear. The reason I say this is because we will spend the rest of this book coding classes, and the more we use them, the clearer they will become.

Furthermore, we have a working bat and a HUD for our Pong game.

In the next chapter, we will code the `Ball` class and get it bouncing around the screen. We will then be able to add collision detection and finish the game.

Frequently asked questions

Q) I have learned other languages and OOP seems much simpler in C++. Is this a correct assessment?

A) This was an introduction to OOP and its basic fundamentals. There is more to it than this. We will learn about more OOP concepts and details throughout this book.

Q) Why do we use the `::` operator in function definitions outside the class declaration?

A) The `::` operator is the scope resolution operator in C++, used to define functions outside the class declaration. When functions are declared inside a class, they are implicitly associated with that class. However, when providing the actual implementation outside the class, we use `ClassName::` before the function name to specify the class to which the function belongs. This ensures correct association and avoids naming conflicts, enhancing code clarity and maintainability.

Q) Should member variables be initialized in the constructor's member initializer list or within the constructor body?

A) It's recommended to initialize member variables in the constructor's member initializer list whenever possible. This approach is more efficient, especially for complex classes, and it ensures that member variables are

initialized before the constructor body is executed. However, simple initialization within the constructor body is perfectly OK when it suits your use, but we should always prefer the member initializer list.

7

AABB Collision Detection and Physics – Finishing the Pong Game

In this chapter, we will code our second class. We will see that although the ball is obviously quite different from the bat, we will use the exact same techniques to encapsulate the appearance and functionality of a ball inside a `Ball` class, just like we did with the bat and the `Bat` class. We will then add the finishing touches to the Pong game by coding some **collision detection** and **scorekeeping**. This might sound complicated but as we are coming to expect, SFML will make things much easier than they otherwise would be.

We will cover the following topics in this chapter:

- Coding the `Ball` class
- Using the `Ball` class
- Collision detection and scoring
- Running the game
- Learning about the C++ spaceship operator

We will start by coding the class that represents the ball.

Coding the Ball class

To get started, we will code the header file. Right-click on **Header Files** in the **Solution Explorer** window and select **ADD | New Item**. Next, choose the **Header File (.h)** option and name the new file `Ball.h`. Then, click the **Add** button. Now, we are ready to code the file.

Add the following code to `Ball.h`:

```
#pragma once
#include <SFML/Graphics.hpp>
using namespace sf;
class Ball
{
private:
    Vector2f m_Position;
    RectangleShape m_Shape;
    float m_Speed = 300.0f;
    float m_DirectionX = .2f;
    float m_DirectionY = .2f;
public:
    Ball(float startX, float startY);
    FloatRect getPosition();
    RectangleShape getShape();
    float getXVelocity();
    void reboundSides();
    void reboundBatOrTop();
    void reboundBottom();
    void update(Time dt);
};
```

The first thing you will notice is the similarity in the member variables compared to the `Bat` class. There is a member variable for the position, appearance, and speed, just like there was for the player's bat, and they are the same types (`Vector2f`, `RectangleShape`, and `float`, respectively). They even have the same names (`m_Position`, `m_Shape`, and `m_Speed`,

respectively). The difference between the member variables of this class is that the direction is handled with two `float` variables that will track horizontal and vertical movement. These are `m_DirectionX` and `m_DirectionY`.

Note that we will need to code eight functions to bring the ball to life. There is a constructor that has the same name as the class, which we will use to initialize a `Ball` instance. There are three functions with the same name and usage as the `Bat` class. They are `getPosition`, `getShape`, and `update`. The `getPosition` and `getShape` functions will share the location and the appearance of the ball with the `main` function, and the `update` function will be called from the `main` function to allow the `Ball` class to update its position once per frame.

The remaining functions control the direction the ball will travel in. The `reboundSides` function will be called from the `main` when a collision is detected with either side of the screen, the `reboundBatOrTop` function will be called in response to the ball hitting the player's bat or the top of the screen, and the `reboundBottom` function will be called when the ball hits the bottom of the screen.

Of course, these are just the declarations, so let's write the C++ that does the work in the `Ball.cpp` file.

Let's create the file, and then we can start discussing the code. Right-click the **Source Files** folder in the **Solution Explorer** window. Now, select **C++ File (.cpp)** and enter `Ball.cpp` in the **Name:** field. Click the **Add** button and our new file will be created for us.

Add the following code to `Ball.cpp`:

```
#include "Ball.h"
// This the constructor function
Ball::Ball(float startX, float startY) : m_Position(startX,star
{
    m_Shape.setSize(sf::Vector2f(10, 10));
    m_Shape.setPosition(m_Position);
}
```

In the preceding code, we have added the required `include` directive for the `Ball` class header file. The constructor function with the same name as the class receives two `float` parameters, which are used to initialize the `m_Position` member's `Vector2f` instance using an initializer list. The `RectangleShape` instance is then sized with the `setSize` function and positioned with `setPosition`. The size that's being used is 10 pixels wide and 10 high; this is arbitrary but works well. The position that's being used is, of course, taken from the `m_Position Vector2f` instance.

Add the following code underneath the constructor in the `Ball.cpp` function:

```
FloatRect Ball::getPosition()
{
    return m_Shape.getGlobalBounds();
}
RectangleShape Ball::getShape()
{
    return m_Shape;
}
float Ball::getXVelocity()
{
    return m_DirectionX;
}
```

In the preceding code, we are coding the three getter functions of the `Ball` class. They each return something to the `main` function. The first, `getPosition`, uses the `getGlobalBounds` function on `m_Shape` to return a `FloatRect` instance. This will be used for collision detection.

The `getShape` function returns `m_Shape` so that it can be drawn in each frame of the game loop. The `getXVelocity` function tells the `main` function which direction the ball is traveling in, and we will see very soon exactly how this is useful to us. Since we don't ever need to get the vertical velocity, there is no corresponding `getYVelocity` function, but it would be simple to add one if we did.

Add the following functions underneath the previous code we just added:

```
void Ball::reboundSides()
{
    m_DirectionX = -m_DirectionX;
}
void Ball::reboundBatOrTop()
{
    m_DirectionY = -m_DirectionY;
}
void Ball::reboundBottom()
{
    m_Position.y = 0;
    m_Position.x = 500;
    m_DirectionY = -m_DirectionY;
}
```

In the preceding code, the three functions whose names begin with `rebound`. handle what happens when the ball collides with various places. In the `reboundSides` function, `m_DirectionX` has its value inverted, which will have the effect of making a positive value negative and a negative value positive, thereby reversing (horizontally) the direction in which the

ball is traveling. `reboundBatOrTop` does the same but with `m_DirectionY`, which has the effect of reversing the direction in which the ball is traveling (vertically). The `reboundBottom` function repositions the ball at the top center of the screen and sends it downward. This is just what we want after the player has missed a ball and it has hit the bottom of the screen.

Finally, for the `Ball` class, add the `update` function as follows:

```
void Ball::update(Time dt)
{
    // Update the ball's position
    m_Position.y += m_DirectionY * m_Speed * dt.asSeconds();
    m_Position.x += m_DirectionX * m_Speed * dt.asSeconds();
    // Move the ball
    m_Shape.setPosition(m_Position);
}
```

In the preceding code, `m_Position.y` and `m_Position.x` are updated using the appropriate direction, velocity, speed, and the amount of time the current frame took to complete. The newly updated `m_Position` values are then used to change the position that the `m_Shape RectangleShape` instance is positioned at. This is the same math that we used for moving the clouds and the bee in the first project. The difference is that this logic is contained within the class. If we ever need to change how the ball moves, it will only affect the code in the `Ball` class.

The `Ball` class is done, so let's put it into action.

Using the Ball class

To put the ball into action, add the following code to make the `Ball` class available in the `main` function:

```
#include "Ball.h"
```

Add the following highlighted line of code to declare and initialize an instance of the `Ball` class using the constructor function that we have just coded:

```
// Create a bat
Bat bat(1920 / 2, 1080 - 20);
// Create a ball
Ball ball(1920 / 2, 0);
// Create a Text object called HUD
Text hud;
Add the following code, positioned exactly as highlighted:
/*
Update the bat, the ball and the HUD
******************************************************
******************************************************
******************************************************
*/
// Update the delta time
Time dt = clock.restart();
bat.update(dt);
ball.update(dt);
// Update the HUD text
std::stringstream ss;
ss << "Score:" << score << "    Lives:" << lives;
hud.setString(ss.str());
```

In the preceding code, we simply call `update` on the `ball` instance. The ball will be repositioned accordingly.

Add the following highlighted code to draw the ball on each frame of the game loop:

```
/*
Draw the bat, the ball and the HUD
```

```
    ********************************************
    ********************************************
    ********************************************
    */
    window.clear();
    window.draw(hud);
    window.draw(bat.getShape());
    window.draw(ball.getShape());
    window.display();
```

At this stage, you could run the game and the ball would spawn at the top of the screen and begin its descent toward the bottom of the screen. It would, however, disappear off the bottom of the screen because we are not detecting any collisions yet. Let's fix that now.

Collision detection and scoring

Unlike in the Timber!!! game, when we simply checked whether a branch in the lowest position was on the same side as the player's character, in this game, we will need to mathematically check for the intersection of the ball with the bat or the ball with any of the four sides of the screen.

Let's look at some hypothetical code that would achieve this so that we understand what we are doing. Then, we will turn to SFML to solve the problem for us.

The code for testing the intersection of two rectangles would look something like this. Don't use the following code. It is for demonstration purposes only:

```
if(objectA.getPosition().right > objectB.getPosition().left
    && objectA.getPosition().left < objectB.getPosition().right
{
```

```
    // objectA is intersecting objectB on x axis
// But they could be at different heights
if(objectA.getPosition().top < objectB.getPosition().bottom
        && objectA.getPosition().bottom > objectB.getPosition()
        {
            // objectA is intersecting objectB
// on y axis as well-
// Collision detected
        }
}
```

The first part of the code tests conditions on the horizontal, or *x*, axis. The first `if` statement is checking whether `objectA` and `objectB` intersect along the horizontal (x) axis. It's comparing the right side of `objectA` (`objectA.getPosition().right`) with the left side of `objectB` (`objectB.getPosition().left`). Additionally, it checks whether the left side of `objectA` is to the left or the right side of `objectB`. If both conditions are true, it means there is an intersection on the *x* axis.

The second part of the code, embedded in the `true` branch of the first part, tests conditions on the vertical or *y* axis. If the first condition is met (i.e., there's an intersection on the x-axis), the code proceeds to the inner `if` statement. Here, it checks whether `objectA` and `objectB` are also intersecting along the vertical (*y*) axis. It compares the top side of `objectA` (`objectA.getPosition().top`) with the bottom side of `objectB` (`objectB.getPosition().bottom`). Furthermore, it checks whether the bottom side of `objectA` is below the top side of `objectB`. If both conditions are true, it means there is an intersection on the y-axis.

Finally, if both the x-axis and y-axis conditions are true, the code inside the innermost block is executed. This block indicates that a collision has been detected between `objectA` and `objectB`. It's a common technique used in

game development to check whether two objects, like game characters or items, are overlapping or colliding in both horizontal and vertical directions.

The technique is called **axis-aligned bounding box** or **AABB** collision detection. This technique is widely used in 2D graphics and game development due to its computational efficiency (i.e., it's fast). It doesn't provide precise collision information for irregularly shaped objects or circles, but even for these types of objects, AABB will often be used as a fast initial check before more mathematically intensive checks are made.

The good news is we don't need to write the preceding code; however, we will be using the SFML `intersects` function, which works on `FloatRect` objects. Think of or look back to the `Bat` and `Ball` classes; they both had a `getPosition` function, which returned a `FloatRect` object representing the object's current location. We will see how we can use `getPosition`, along with `intersects`, to do all our collision detection.

Add the following highlighted code at the end of the update section of the `main` function:

```
/*
Update the bat, the ball and the HUD
****************************************
****************************************
****************************************
*/
// Update the delta time
Time dt = clock.restart();
bat.update(dt);
ball.update(dt);
// Update the HUD text
std::stringstream ss;
ss << "Score:" << score << "    Lives:" << lives;
hud.setString(ss.str());
```

```
// Handle ball hitting the bottom
if (ball.getPosition().top > window.getSize().y)
{
 // reverse the ball direction
    ball.reboundBottom();
 // Remove a life
    lives--;
 // Check for zero lives
 if (lives < 1) {
 // reset the score
        score = 0;
 // reset the lives
        lives = 3;
    }
}
```

In the preceding code, the first `if` condition checks whether the ball has hit the bottom of the screen:

```
if (ball.getPosition().top > window.getSize().y)
```

If the top of the ball is at a greater position than the height of the window, then the ball has disappeared off the bottom of the player's view. In response, the `ball.reboundBottom` function is called. Remember that, in this function, the ball is repositioned at the top of the screen. At this point, the player has lost a life, so the `lives` variable is decremented.

The second `if` condition checks whether the player has run out of lives (`lives < 1`). If this is the case, the score is reset to 0, the number of lives is reset to 3, and the game is restarted. In the next project, we will learn how to keep and display the player's highest score.

Add the following code underneath the previous code:

```
// Handle ball hitting top
if (ball.getPosition().top < 0)
{
    ball.reboundBatOrTop();
    // Add a point to the players score
    score++;
}
```

In the preceding code, we are detecting that the top of the ball hits the top of the screen. When this occurs, the player is awarded a point and the `ball.reboundBatOrTop` function is called, which reverses the vertical direction of travel and sends the ball back toward the bottom of the screen.

Add the following code underneath the previous code:

```
// Handle ball hitting sides
if (ball.getPosition().left < 0 ||
    ball.getPosition().left + ball.getPosition().width> window.
{
    ball.reboundSides();
}
```

In the preceding code, the `if` condition detects a collision with the **left-hand side** (**LHS**) of the ball and the LHS of the screen or the **right-hand side** (**RHS**) of the ball (left + 10) with the RHS of the screen. In either event, the `ball.reboundSides` function is called and the horizontal direction of travel is reversed.

Add the following code:

```
// Has the ball hit the bat?
if (ball.getPosition().intersects(bat.getPosition()))
{
    // Hit detected so reverse the ball and score a point
```

```
        ball.reboundBatOrTop();
    }
```

In the preceding code, the `intersects` function is used to determine whether the ball has hit the bat. When this occurs, we use the same function that we used for a collision with the top of the screen to reverse the vertical direction of travel of the ball.

Running the game

You can now run the game and bounce the ball around the screen. The score will increase when you hit the ball with the bat and the lives will decrease when you miss it. When `lives` reaches 0, the score will reset, and the `lives` variable will go back up to 3, as follows:

Figure 7.1: Running the game

Learning about the C++ spaceship operator

As this is a short chapter, I thought it would be a good place to learn some more C++. We don't need this theory in the current project. Yes, the **spaceship operator** is a real thing. It is another neat C++ operator.

The spaceship operator, represented by `<=>`, is a relatively new addition to the C++ language, introduced in **C++20**. It is used for three-way comparisons between two objects, which means it helps determine whether one object is less than, equal to, or greater than another. The spaceship operator returns one of three values: <, ==, or >, indicating the relationship between the two objects. Here's how it works.

If the LHS of the spaceship operator is less than the RHS, it returns a negative value. This indicates that the LHS is "less than" the RHS.

If the LHS is equal to the RHS, it returns 0. This indicates that the two objects are equal.

If the LHS is greater than the RHS, it returns a positive value. This indicates that the LHS is "greater than" the RHS. An example will help.

```
int a = 5;
int b = 10;
//  Next we use the spaceship operator
int result = a <=> b;
 if (result < 0)
{
 // a is less than b
 }
else if (result == 0)
{
// a is equal to b
```

```
 }
else if (result > 0)
{
 // a is greater than b
 }
```

In the preceding code, we declare two integers, `a` and `b`, and then we use the spaceship operator, `<=>`, to compare them. The result of the comparison is stored in the `int` variable `result`. Remember that the returned value will be negative, zero, or positive, indicating `a` is less than, equal to, or more than `b`, respectively.

We then check the value of the `result` to determine the relationship between `a` and `b` and respond differently depending on the `result`.

In fact, this simplified code example hides some additional learning information. The result of using the spaceship operator actually returns a new C++ type called `strong_ordering`. Fortunately, **strong_ordering** is convertible to `int`. The **strong_ordering** type represents the result of a three-way comparison.

Summary

Congratulations: that's the second game completed! We could have added more features to that game, such as co-op play, high scores, and sound effects, but I just wanted to use the simplest possible example to introduce classes and AABB collision detection. Now that we have these topics in our game developer's arsenal, we can move on to a much more exciting project and even more game development topics.

In the next chapter, we will plan the Zombie Arena game, learn about the SFML `View` class, which acts as a virtual camera in our game world, and

code some more classes.

Frequently asked questions

Q) Isn't this game a little quiet?

A) I didn't add sound effects to this game because I wanted to keep the code as short as possible while using our first classes and learning to use the time to smoothly animate all the game objects. If you want to add sound effects, then all you need to do is add the `.wav` files to the project, use SFML to load the sounds, and play a sound effect in each of the collision events. We will have sound in the next project.

Q) The game is too easy! How can I make the ball speed up a little?

A) There are lots of ways you can make the game more challenging. One simple way would be to add a line of code in the `Ball` class' `reboundBatOrTop` function that increases the speed. As an example, the following code would increase the speed of the ball by 10% each time the function is called:

```
// Speed up a little bit on each hit
m_Speed = m_Speed * 1.1f;
```

The ball would get quite fast quite quickly. You would then need to devise a way to reset the speed back to `300.0f` when the player has lost all their lives. You could create a new function in the `Ball` class, perhaps called `resetSpeed`, and call it from `main` when the code detects that the player has lost their last life.

Q) Name one advantage and one disadvantage of AABB collision detection.

A) AABB is computationally efficient, making it suitable for demanding applications like games where frequent collision checks are necessary. It is also simple to understand, but that’s two advantages. While AABB is efficient, it can’t provide precise collision information for irregularly shaped objects.

8

SFML Views – Starting the Zombie Shooter Game

In this project, we will be making even more use of **OOP** and to a powerful effect. We will also be exploring the SFML `View` class. This versatile class will allow us to easily divide our game into layers for different aspects of the game. In the Zombie Shooter project, we will have a layer for the **heads-up display** (**HUD**) and a layer for the main game. This is necessary because the game world expands each time the player clears a wave of zombies. Eventually, the game world will be bigger than the screen and will need to scroll. The use of the **View** class will prevent the text of the HUD from scrolling with the background.

This is what we will cover in this chapter:

- Planning and starting the Zombie Arena game
- OOP and the Zombie Arena project
- Building the player – the first class
- Controlling the game camera with SFML View
- Starting the Zombie Arena game engine
- Managing the code files
- Starting to code the main game loop

You will find this chapter's source code in the GitHub repository: https://github.com/PacktPublishing/Beginning-C-Game-Programming-Third-Edition/tree/main/ZombieShooter

Planning and starting the Zombie Arena game

At this point, if you haven't already, I suggest you go and watch a video of *Over 9000 Zombies* (http://store.steampowered.com/app/273500/) and *Crimson Land* (http://store.steampowered.com/app/262830/). Our game will obviously not be as in depth or advanced as either of these examples, but we will also have the same basic set of features and game mechanics, such as the following:

- A HUD that shows details such as the score, high score, bullets in the clip, the number of bullets left, player health, and the number of zombies left to kill.
- The player will shoot zombies while frantically running away from them.
- Move around a scrolling world using the *WASD* keyboard keys while aiming the gun using the mouse.
- In between each level, the player will choose a "level-up" that will affect the way the game needs to be played for the player to win.
- The player will need to collect "pick-ups" to restore health and ammunition.

- Each wave brings more zombies and a bigger arena to make it more challenging.

There will be three types of zombies to splatter. They will have different attributes, such as appearance, health, and speed. We will call them chasers, bloaters, and crawlers. One will be fast, one will be fat, and one will crawl along the floor, respectively. Take a look at the following annotated screenshot of the game to see some of the features in action and the components and assets that make up the game:

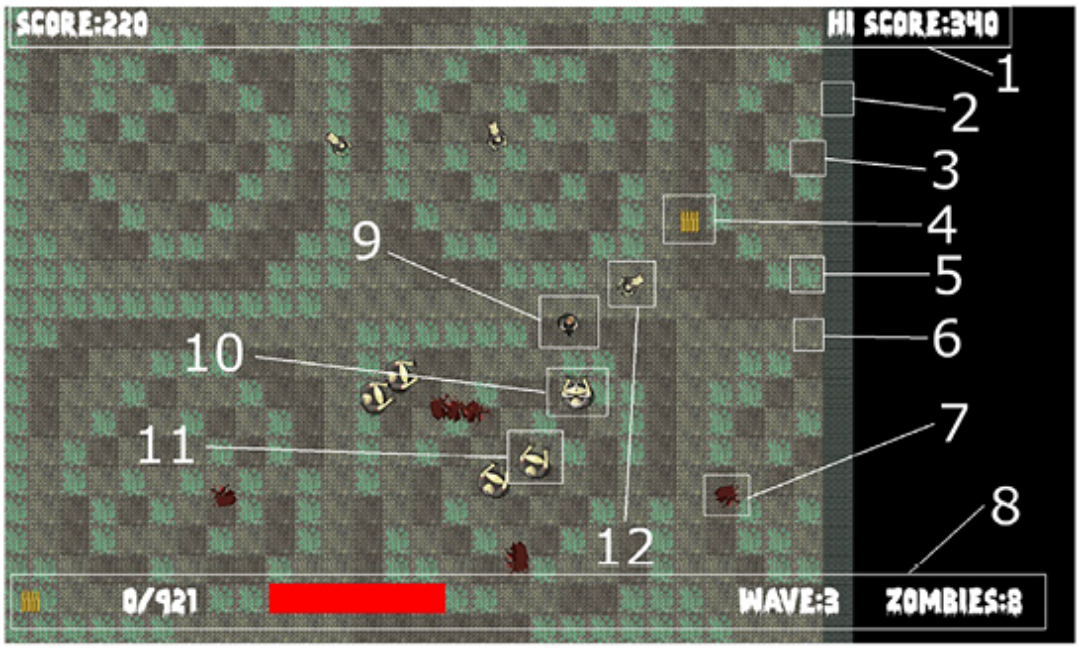


Figure 8.1: Features in action, the components and assets that make up the game

Here is some more information about each of the numbered points:

- The score and hi-score: These, along with the other parts of the HUD, will be drawn in a separate layer, known as a view, and represented by an instance of the `View` class. The hi-score will be saved and loaded to a file.

- A texture that will build a wall around the arena: This texture is contained in a single graphic called a **sprite sheet**, along with the other background textures (points **3**, **5**, and **6**).
- The first of two mud textures from the sprite sheet.
- This is an "ammo pick-up." When the player gets this, they will be given more ammunition. There is a "health pick-up" as well, from which the player will receive more health. These pick-ups can be chosen by the player to be upgraded in between waves of zombies.
- A grass texture, also from the sprite sheet.
- The second mud texture, from the sprite sheet.
- A blood splat where there used to be a zombie.
- The bottom part of the HUD: From left to right, there is an icon to represent ammo, the number of bullets in the clip, the number of spare bullets, a health bar, the current wave of zombies, and the number of zombies remaining for the current wave.
- The player's character.
- A crosshair, which the player aims with the mouse.
- A slow-moving, but strong, "bloater" zombie.
- A slightly faster-moving, but weaker, "crawler" zombie. There is also a "chaser zombie" that is very fast and weak. Unfortunately, I couldn't manage to get one in the screenshot before they were all killed.

So, we have a lot to do and new C++ skills to learn. Let's start by creating a new project.

Creating a new project

As setting up a project is a fiddly process, we will go through it step by step as we did for the Timber!!! project. I won't show you the same images as I

did for the Timber!!! project, but the process is the same; so, flip back to *Chapter 1*, *C++, SFML, Visual Studio, and Starting the First Game*, if you want a reminder of the locations of the various project properties.

Let's run through the following steps:

1. Start Visual Studio and click on the **Create New Project** button. If you have another project open, you can select **File | New project**.
2. In the window shown next, choose **Console app** and click on the **Next** button. You will then see the **Configure your new project** window.
3. In the **Configure your new project** window, type `Zombie Arena` in the **Project name** field.
4. In the **Location** field, browse to the `VS Projects` folder.
5. Check the **Place solution and project in the same directory** option.
6. When you have completed the preceding steps, click on **Create**.
7. We will now configure the project to use the SFML files that we put in the `SFML` folder. From the main menu, select **Project | Zombie Arena properties…**. At this stage, you should have the **Zombie Arena Property Pages** window open.
8. In the **Zombie Arena Property Pages** window, select **All Configurations** from the **Configuration:** drop-down menu, and make sure the drop-down menu to the right is set to **Win32**, not **x64**.
9. Now, select **C/C++** and then **General** from the left-hand menu.
10. Next, locate the **Additional Include Directories** edit box and type the drive letter where your `SFML` folder is located, followed by `\SFML\include.` The full path to type, if you located your `SFML` folder on your D: drive, will be `D:\SFML\include`. Vary your path if you installed SFML on a different drive.
11. Click on **Apply** to save your configurations so far.

12. Now, still in the same window, perform these next steps. From the left-hand menu, select **Linker** and then **General**.
13. Now, find the **Additional Library Directories** edit box and type the drive letter where your `SFML` folder is, followed by `\SFML\lib`. So, the full path to type, if you located your `SFML` folder on your D drive, will be `D:\SFML\lib`. Change your path if you installed SFML on a different drive.
14. Click on **Apply** to save your configurations so far.
15. Select **Linker** and then **Input**.
16. Find the **Additional Dependencies** edit box and click on it on the far left-hand side. Now, copy and paste/type the following: `sfml-graphics-d.lib;sfml-window-d.lib;sfml-system-d.lib;sfml-network-d.lib;sfml-audio-d.lib;`. Be extra careful to place the cursor exactly at the start of the edit box's current content so as not to overwrite any of the text that is already there.
17. Click on **OK**.
18. Click on **Apply** and then **OK**.
19. Back on the main Visual Studio screen, check the main menu toolbar is set to **Debug** and **x86**, not **x64**.

Now, you have configured the project properties and you are nearly ready to go. Next, we need to copy the SFML `.dll` files into the main project directory by following these steps:

1. My main project directory is `D:\VS Projects\Zombie Arena`. This folder was created by Visual Studio in the previous steps. If you put your `Projects` folder somewhere else, then perform this step in your directory. The files we need to copy into the project folder are in your

`SFML\bin` folder. Open a window for each of the two locations and highlight all the `.dll` files.

2. Now, copy and paste the highlighted files into the project.

The project is now set up and ready to go. Next, we will explore and add the project assets.

The project assets

The assets in this project are more numerous and more diverse than the previous games. The assets include the following:

- A font for the text on the screen
- Sound effects for different actions such as shooting, reloading, or getting hit by a zombie
- Graphics for the character, zombies, and a sprite sheet for the various background textures

All the graphics and sound effects that are required for the game are included in the download bundle. They can be found in the `Chapter 8/graphics` and `Chapter 8/sound` folders, respectively.

The font that is required has not been supplied. This is done to avoid any possible ambiguity regarding the license. This will not cause a problem because the links for downloading the fonts and how and where to choose the font will be provided.

Exploring the assets

The graphical assets make up the parts of the scene of our Zombie Arena game. Look at the following graphical assets; it should be clear to you where the assets in the game will be used:

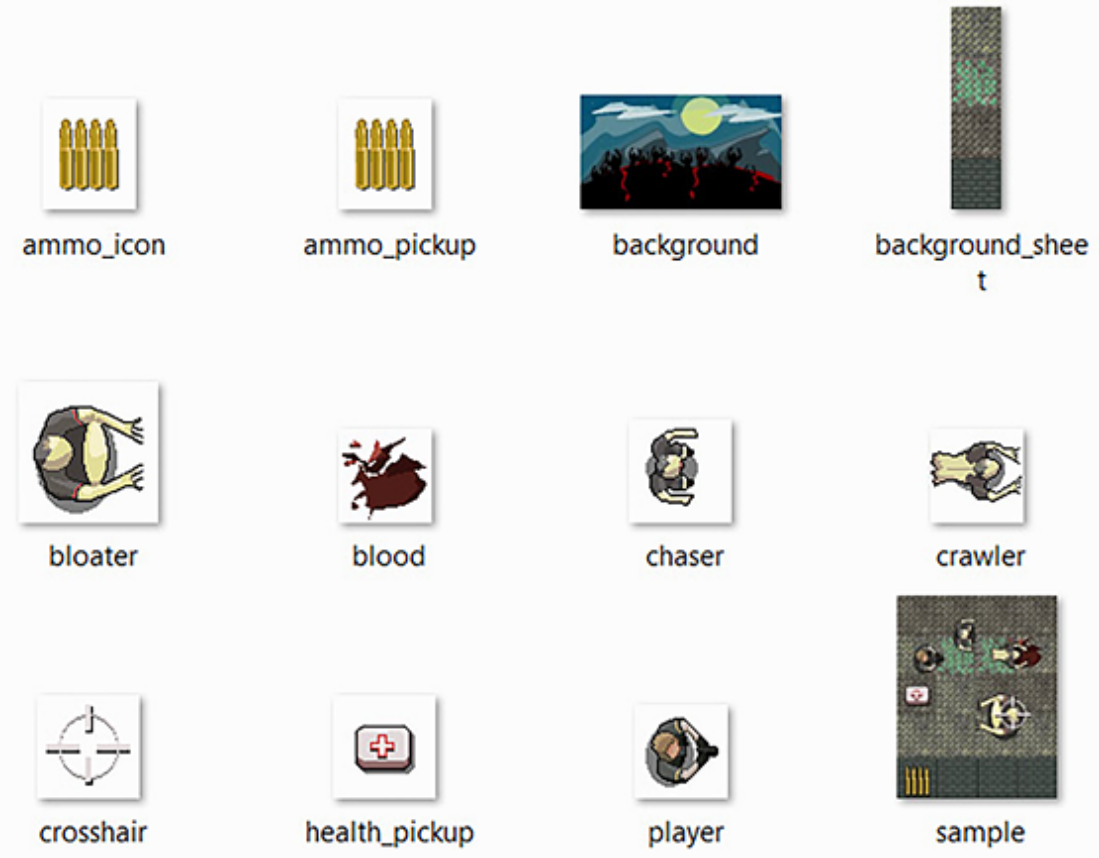


Figure 8.2 Graphical assets

What might be less obvious, however, is the `background_sheet.png` file, which contains four different images. This is the sprite sheet we mentioned previously. We will see how we can save memory and increase the speed of our game using the sprite sheet in *Chapter 9*, *C++ References, Sprite Sheets, and Vertex Arrays*.

The sound files are all in `.wav` format. These are files that contain the sound effects that will be played when certain events are triggered. They are as follows:

- `hit.wav`: A sound that plays when a zombie comes into contact with the player.

- `pickup.wav`: A sound that plays when the player collides or steps on (collects) a health boost (pick-up).
- `powerup.wav`: A sound for when the player chooses an attribute to increase their strength (power-up) in between each wave of zombies.
- `reload.wav`: A satisfying click to let the player know they have loaded a fresh clip of ammunition.
- `reload_failed.wav`: A less satisfying sound that indicates failing to load new bullets.
- `shoot.wav`: A shooting sound.
- `splat.wav`: A sound like a zombie being hit by a bullet.

Once you have decided which assets you will use, it is time to add them to the project.

Adding the assets to the project

The following instructions will assume you are using all the assets that were supplied in the book's download bundle. Where you are using your own assets, simply replace the appropriate sound or graphic file with your own, using the same filename. Let's take a look at the steps:

1. Browse to `D:\VS Projects\ZombieArena`.
2. Create three new folders within this folder and name them `graphics`, `sound`, and `fonts`.
3. From the download bundle, copy the entire contents of `Chapter 8/graphics` into the `D:\VS Projects\ZombieArena\graphics` folder.
4. From the download bundle, copy the entire contents of `Chapter 8/sound` into the `D:\VS Projects\ZombieArena\sound` folder.

5. Now, visit `http://www.1001freefonts.com/zombie_control.font` in your web browser and download the **Zombie Control** font.
6. Extract the contents of the zipped download and add the `zombiecontrol.ttf` file to the `D:\VS Projects\ZombieArena\fonts` folder.

Now, it's time to consider how OOP will help us with this project and then we can start writing the code for Zombie Arena.

OOP and the Zombie Arena project

The initial problem we are facing is the complexity of the current project. Let's consider that there is just a single zombie; here is what we need to make it function in the game:

- Its horizontal and vertical position
- Its size
- The direction it is facing
- A different texture for each zombie type
- A sprite
- A different speed for each zombie type
- A different health for each zombie type
- Keeping track of the type of each zombie
- Collision detection data
- Its intelligence (to chase the player), which is slightly different for each type of zombie
- An indication of whether the zombie is alive or dead

This perhaps suggests a dozen variables for just one zombie, and entire arrays of each of these variables will be required for managing a zombie horde. But what about all the bullets from the machine gun, the pick-ups, and the different level-ups? The code from the much simpler Timber!!! and Pong games also started to get a bit unmanageable, and it is easy to speculate that this more complicated shooter will be many times worse!

Fortunately, we will put all the OOP skills we learned in the previous two chapters into action, as well as learn some new C++ techniques.

We will start our coding for this project with a class to represent the player.

Building the player - the first class

Let's think about what our `Player` class will need to do and what we require for it. The class will need to know how fast it can move, where in the game world it currently is, and how much health it has. As the `Player` class, in the player's eyes, is represented as a 2D graphical character, the class will need both a `Sprite` object and a `Texture` object.

Furthermore, although the reasons might not be obvious at this point, our `Player` class will also benefit from knowing a few details about the overall environment the game is running in. These details are screen resolution, the size of the tiles that make up an arena, and the overall size of the current arena.

As the `Player` class will be taking full responsibility for updating itself in each frame (like the bat and ball did), it will need to know the player's intentions at any given moment. For example, is the player currently holding down a keyboard direction key? Or, is the player currently holding down

multiple keyboard direction keys? Boolean variables will be essentially used to determine the status of the *W*, *A*, *S*, and *D* keys.

It is clear that we are going to need quite a selection of variables in our new class. Having learned all we have about OOP, we will, of course, be making all of these variables private. This means that we must provide access, where appropriate, from the `main` function.

We will use a whole bunch of getter functions as well as some functions to set up our object. These functions are quite numerous. There are 21 functions in this class.

At first, this might seem a little daunting, but we will go through all of them and see that most of them simply set or get one of the private variables.

There are just a few in-depth functions: `update`, which will be called once each frame from the `main` function, and `spawn`, which will handle initializing some of the private variables each time the player is spawned. We will see, however, there is nothing complicated.

The best way to proceed is to code the header file. This will give us the opportunity to see all the private variables and examine all the function signatures.

Pay close attention to the return values and argument types, as this will make understanding the code in the function definitions much easier.

Coding the Player class header file

Start by right-clicking on **Header Files** in **Solution Explorer** and select **Add | New Item…**. In the **Add New Item** window, highlight (by left-

clicking on it) the **Header File (.h)**, and then, in the **Name** field, type `Player.h`. Finally, click on the **Add** button. We are now ready to code the header file for our first class.

Start coding the `Player` class by adding the declaration, including the opening and closing curly braces, followed by a semicolon:

```
#pragma once
#include <SFML/Graphics.hpp>
using namespace sf;
class Player
{
};
```

Now, let's add all our private member variables to the file. Based on what we have already discussed, see whether you can work out what each of them will do. We will go through them individually in a moment:

```
class Player
{
private:
 const float START_SPEED = 200;
 const float START_HEALTH = 100;
 // Where is the player
    Vector2f m_Position;
 // The sprite
    Sprite m_Sprite;
 // And a texture
 // !!Watch this space - Changes here soon!!
    Texture m_Texture;
 // What is the screen resolution
    Vector2f m_Resolution;
 // What size is the current arena
    IntRect m_Arena;
 // How big is each tile of the arena
 int m_TileSize;
 // Which direction(s)the player is moving in
```

```
    bool m_UpPressed;
    bool m_DownPressed;
    bool m_LeftPressed;
    bool m_RightPressed;
    // How much health has the player got?
    int m_Health;
    // What is the max' health the player can have
    int m_MaxHealth;
    // When was the player last hit
        Time m_LastHit;
    // Speed in pixels per second
    float m_Speed;
    // All our public functions next
};
```

The previous code declares all our member variables. Some are regular variables, while some of them are objects. Notice that they are all under the `private` section of the class and, therefore, are not directly accessible from outside the class.

Also, notice that we are using the naming convention of prefixing `m_` to all the names of the non-constant variables. The `m_` prefix will remind us, while coding the function definitions, that they are member variables, distinct from the local variables we will create in some of the functions, and also distinct from the function parameters.

All the variables that are used are straightforward, such as `m_Position`, `m_Texture`, and `m_Sprite`, which are for the current location, texture, and sprite of the player, respectively. In addition to this, each variable (or group of variables) is commented on to make its usage plain.

However, why exactly they are needed, and the context they will be used in, might not be so obvious. For example, `m_LastHit`, which is an object of the `Time` type, is for recording the time that the player last received a hit from a

zombie. It is not obvious why we might need this information, but we will go over this soon.

As we piece the rest of the game together, the context for each of the variables will become clearer. The important thing, for now, is to familiarize yourself with the names and data types to make following along with the rest of the project trouble free.

You don't need to memorize the variable names and types as we will discuss all the code when they are used. You do, however, need to take your time to look over them and get more familiar with them. Furthermore, as we proceed, it might be worth referring to this header file if anything seems unclear.

Now, we can add a complete long list of functions. Add the following highlighted code and see whether you can work out what it all does. Pay close attention to the return types, parameters, and the name of each function. This is key to understanding the code we will write throughout the rest of this project. What do they tell us about each function? Add the following highlighted code and then we will examine it:

```
// All our public functions next
public:
 Player();
 void spawn(IntRect arena, Vector2f resolution, int tileSize);
 // Call this at the end of every game
 void resetPlayerStats();

 // Handle the player getting hit by a zombie
 bool hit(Time timeHit);
 // How long ago was the player last hit
```

```
    Time getLastHitTime();
    // Where is the player
    FloatRect getPosition();
    // Where is the center of the player
    Vector2f getCenter();
    // What angle is the player facing
    float getRotation();
    // Send a copy of the sprite to the main function
    Sprite getSprite();
    // The next four functions move the player
    void moveLeft();
    void moveRight();
    void moveUp();
    void moveDown();
    // Stop the player moving in a specific direction
    void stopLeft();
    void stopRight();
    void stopUp();
    void stopDown();
    // We will call this function once every frame
    void update(float elapsedTime, Vector2i mousePosition);
    // Give the player a speed boost
    void upgradeSpeed();
    // Give the player some health
    void upgradeHealth();
    // Increase the max' health the player can have
    void increaseHealthLevel(int amount);
    // How much health has the player currently got?
    int getHealth();
};
```

Firstly, note that all the functions are public. This means we can call all these functions using an instance of the class from the `main` function with code like this:

```
player.getSprite();
```

Assuming `player` is a fully set-up instance of the `Player` class, the previous code will return a copy of `m_Sprite`. Putting this code into a real context, we could, in the `main` function, write code like this:

```
window.draw(player.getSprite());
```

The previous code would draw the player graphic in its correct location, just as if the sprite was declared in the `main` function itself. This is what we did with the `Bat` class in the Pong project.

Before we move on to implementing (that is, writing the definitions) these functions in a corresponding `.cpp` file, let's take a closer look at each of them in turn:

- `void spawn(IntRect arena, Vector2f resolution, int tileSize)`: This function does what its name suggests. It will prepare the object ready for use, which includes putting it in its starting location (that is, spawning it). Notice that it doesn't return any data, but it does have three arguments. It receives an `IntRect` instance called `arena`, which will be the size and location of the current level; a `Vector2f` instance, which will contain the screen resolution; and an `int`, which will hold the size of a background tile.
- `void resetPlayerStats`: Once we give the player the ability to level up between waves, we will need to be able to take away/reset those abilities at the start of a new game.
- `Time getLastHitTime()`: This function does just one thing – it returns the time when the player was last hit by a zombie. We will use this function when detecting collisions, and it will allow us to make sure that the player isn't punished *too* frequently for making contact with a zombie.

- `FloatRect getPosition()`: This function returns a `FloatRect` instance that describes the horizontal and vertical floating-point coordinates of the rectangle, which contains the player graphic. This is also useful for collision detection.
- `Vector2f getCenter()`: This is slightly different from `getPosition` because it is a `Vector2f` type and contains just the `x` and `y` locations of the very center of the player graphic.
- `float getRotation()`: The code in the `main` function will sometimes need to know, in degrees, which way the player is currently facing. 3 o'clock is 0 degrees and increases clockwise.
- `Sprite getSprite()`: As we discussed previously, this function returns a copy of the sprite that represents the player.
- `void moveLeft(), ..Right(), ..Up(), ..Down()`: These four functions have no return type or parameters. They will be called from the `main` function and the `Player` class will then be able to act when one or more of the *WASD* keys have been pressed.
- `void stopLeft(), ..Right(), ..Up(), ..Down()`: These four functions have no return type or parameters. They will be called from the `main` function, and the `Player` class will then be able to act when one or more of the *WASD* keys have been released.
- `void update(float elapsedTime, Vector2i mousePosition)`: This will be the only long function of the entire class. It will be called once per frame from `main`. It will do everything necessary to make sure the `player` object's data is updated so that it's ready for collision detection and drawing. Notice that it returns no data but receives the amount of elapsed time since the last frame, along with a `Vector2i` instance, which will hold the horizontal and vertical screen location of the mouse pointer/crosshair.

Note that these are integer screen coordinates and are distinct from the floating-point world coordinates.

- `void upgradeSpeed()`: A function that can be called from the leveling-up screen when the player chooses to make the character move faster.
- `void upgradeHealth()`: Another function that can be called from the leveling-up screen when the player chooses to make the character move stronger (that is, have more health).
- `void increaseHealthLevel(int amount)`: A subtle but important difference regarding the previous function is that this one will increase the amount of health the player has, up to the maximum that's currently set. This function will be used when the player picks up a health pick-up.
- `int getHealth()`: With the level of health being as dynamic as it is, we need to be able to determine how much health the player has at any given moment. This function returns an `int`, which holds that value.

Like the variables, it should now be plain what each of the functions is for. Also, the *why* and the precise context of using some of these functions will only reveal themselves as we progress with the project.

You don't need to memorize the function names, return types, or parameters as we will discuss the code when they are used. You do, however, need to take your time to look over them, along with the previous explanations, and get more familiar with them. Furthermore, as we proceed, it might be worth referring to this header file if anything seems unclear.

Now, we can move on to the meat of our functions: the definitions.

Coding the Player class function definitions

Finally, we can begin writing the code that does the work of our class.

Right-click on **Source Files** in **Solution Explorer** and select **Add | New Item**. In the **Add New Item** window, highlight (by left-clicking on) **C++ File (.cpp)** and then, in the **Name** field, type `Player.cpp`. Finally, click on the **Add** button.

From now on, I will simply ask you to create a new class or header file. So, commit the preceding step to memory or refer back here if you need a reminder.

We are now ready to code the `.cpp` file for our first class in this project.

Here are the necessary `include` directives, followed by the definition of the constructor. Remember, the constructor will be called when we first instantiate an object of the `Player` type. Add the following code to the `Player.cpp` file and then we can take a closer look at it:

```
#include "player.h"
Player::Player()
    : m_Speed(START_SPEED),
      m_Health(START_HEALTH),
      m_MaxHealth(START_HEALTH),
      m_Texture(),
      m_Sprite()
{
    // Associate a texture with the sprite
// !!Watch this space!!
```

```
    m_Texture.loadFromFile("graphics/player.png");
    m_Sprite.setTexture(m_Texture);
    // Set the origin of the sprite to the center,
// for smooth rotation
    m_Sprite.setOrigin(25, 25);
}
```

In the constructor function, which, of course, has the same name as the class and no return type, we write code that begins to set up the `Player` object so that it is ready for use.

To be clear, this code will run when we write the following code from the `main` function:

```
Player player;
```

Don't add the previous line of code just yet.

The `m_Speed`, `m_Health`, `m_MaxHealth`, `m_Texture`, and `m_Sprite` members are initialized in the initializer list. This is considered a good practice as it can lead to more efficient code and ensures that members are initialized before entering the constructor body.

All we do in the constructor body is load the player graphic into `m_Texture`, associate `m_Texture` with `m_Sprite`, and set the origin of `m_Sprite` to the center, `(25, 25)`.

Note the cryptic comment, `// !!Watch this space!!`, indicating that we will return to the loading of our texture and some important issues regarding it. We will eventually change how we deal with this texture once we have discovered a problem and learned a bit more about C++. We

> will do so in *Chapter 10*, *Pointers, the Standard Template Library, and Texture Management*.

Next, we will code the `spawn` function. We will only ever create one instance of the `Player` class. We will, however, need to spawn it into the current level for each wave. This is what the `spawn` function will handle for us. Add the following code to the `Player.cpp` file and be sure to examine the details and read the comments:

```
void Player::spawn(IntRect arena,
        Vector2f resolution,
 int tileSize)
{
    // Place the player in the middle of the arena
    m_Position.x = arena.width / 2;
    m_Position.y = arena.height / 2;
    // Copy the details of the arena
// to the player's m_Arena
    m_Arena.left = arena.left;
    m_Arena.width = arena.width;
    m_Arena.top = arena.top;
    m_Arena.height = arena.height;
    // Remember how big the tiles are in this arena
    m_TileSize = tileSize;
    // Store the resolution for future use
    m_Resolution.x = resolution.x;
    m_Resolution.y = resolution.y;
}
```

The preceding code starts by initializing the `m_Position.x` and `m_Position.y` values to half the height and width of the passed-in `arena`. This has the effect of moving the player to the center of the level, regardless of its size.

Next, we copy all the coordinates and dimensions of the passed-in `arena` to the member object of the same type, `m_Arena`. The details of the size and coordinates of the current arena are used so frequently that it makes sense to do this. We can now use `m_Arena` for tasks such as making sure the player can't walk through walls. In addition to this, we copy the passed in `tileSize` instance to the member variable, `m_TileSize`, for the same purpose. We will see `m_Arena` and `m_TileSize` in action in the `update` function.

The final two lines from the preceding code copy the screen resolution from the Vector2f resolution, which is a parameter of `spawn`, into `m_Resolution`, which is a member variable of `Player`. We now have access to these values inside the `Player` class.

Now, add the very straightforward code of the `resetPlayerStats` function:

```
void Player::resetPlayerStats()
{
    m_Speed = START_SPEED;
    m_Health = START_HEALTH;
    m_MaxHealth = START_HEALTH;
}
```

When the player dies, we will use this to reset any upgrades they might have used.

We will not write the code that calls the `resetPlayerStats` function until we have nearly completed the project, but it is there ready for when we need it.

In the next part of the code, we will add two more functions. They will handle what happens when the player is hit by a zombie. We will be able to call `player.hit()` and pass it in the current game time. We will also be able to query the last time that the player was hit by calling

`player.getLastHitTime()`. Exactly how these functions are useful will become apparent when we have some zombies.

Add the two new definitions to the `Player.cpp` file and then we will examine the C++ code a little more closely:

```
Time Player::getLastHitTime()
{
    return m_LastHit;
}
bool Player::hit(Time timeHit)
{
    if (timeHit.asMilliseconds()
        - m_LastHit.asMilliseconds() > 200)
    {
        m_LastHit = timeHit;
        m_Health -= 10;
        return true;
    }
    else
    {
        return false;
    }
}
```

The code for `getLastHitTime()` is very straightforward; it will return whatever value is stored in `m_LastHit`.

The `hit` function is a bit more in depth and nuanced. First, the `if` statement checks to see whether the time that's passed in as a parameter is 200 milliseconds further ahead than the time stored in `m_LastHit`. If it is, `m_LastHit` is updated with the time passed in and `m_Health` has 10 deducted from its current value. The last line of code in this `if` statement is `return true`. Notice that the `else` clause simply returns `false` to the calling code.

The overall effect of this function is that health points will only be deducted from the player up to five times per second. Remember that our game loop might be running at thousands of iterations per second. In this scenario, without the restriction this function provides, a zombie would only need to be in contact with the player for one second, and tens of thousands of health points would be deducted. The `hit` function controls and restricts this phenomenon. It also lets the calling code know whether a new hit has been registered (or not) by returning `true` or `false`.

This code implies that we will detect collisions between a zombie and the player in the `main` function. We will then call `player.hit()` to determine whether to deduct any health points.

Next, for the `Player` class, we will implement a bunch of getter functions. They allow us to keep the data neatly encapsulated in the `Player` class at the same time as their values are being made available to the `main` function.

Add the following code right after the previous block:

```
FloatRect Player::getPosition()
{
    return m_Sprite.getGlobalBounds();
}
Vector2f Player::getCenter()
{
    return m_Position;
}
float Player::getRotation()
{
    return m_Sprite.getRotation();
}
Sprite Player::getSprite()
{
    return m_Sprite;
}
int Player::getHealth()
```

```
{
    return m_Health;
}
```

The preceding code is very straightforward. Each of the previous five functions returns the value of one of our member variables. Look carefully at each of them and familiarize yourself with which function returns which value.

The next eight short functions enable the keyboard controls (which we will use from the `main` function) so that we can change the data contained in our object of the `Player` type. Add the following code to the `Player.cpp` file and then we will summarize how it all works:

```
void Player::moveLeft()
{
    m_LeftPressed = true;
}
void Player::moveRight()
{
    m_RightPressed = true;
}
void Player::moveUp()
{
    m_UpPressed = true;
}
void Player::moveDown()
{
    m_DownPressed = true;
}
void Player::stopLeft()
{
    m_LeftPressed = false;
}
void Player::stopRight()
{
    m_RightPressed = false;
}
```

```
void Player::stopUp()
{
    m_UpPressed = false;
}
void Player::stopDown()
{
    m_DownPressed = false;
}
```

The previous code has four functions (`moveLeft`, `moveRight`, `moveUp`, and `moveDown`), which set the related Boolean variables (`m_LeftPressed`, `m_RightPressed`, `m_UpPressed`, and `m_DownPressed`) to `true`. The other four functions (`stopLeft`, `stopRight`, `stopUp`, and `stopDown`) do the opposite and set the same Boolean variables to `false`. The instance of the `Player` class can now be kept informed of which of the *WASD* keys were pressed and which were not.

The following function is the one that does all the hard work. The `update` function will be called once in every single frame of our game loop. Add the following code, and then we will examine it in detail. If you followed along with the previous eight functions and we remember how we animated the clouds and bees for the Timber!!! project and the bat and ball for Pong, you will probably understand most of the following code:

```
void Player::update(float elapsedTime, Vector2i mousePosition)
{
    if (m_UpPressed)
    {
        m_Position.y -= m_Speed * elapsedTime;
    }
    if (m_DownPressed)
    {
        m_Position.y += m_Speed * elapsedTime;
    }
    if (m_RightPressed)
```

```
        {
            m_Position.x += m_Speed * elapsedTime;
        }
        if (m_LeftPressed)
        {
            m_Position.x -= m_Speed * elapsedTime;
        }
        m_Sprite.setPosition(m_Position);
        // Keep the player in the arena
    if (m_Position.x > m_Arena.width - m_TileSize)
        {
            m_Position.x = m_Arena.width - m_TileSize;
        }
        if (m_Position.x < m_Arena.left + m_TileSize)
        {
            m_Position.x = m_Arena.left + m_TileSize;
        }
        if (m_Position.y > m_Arena.height - m_TileSize)
        {
            m_Position.y = m_Arena.height - m_TileSize;
        }
        if (m_Position.y < m_Arena.top + m_TileSize)
        {
            m_Position.y = m_Arena.top + m_TileSize;
        }
        // Calculate the angle the player is facing
    float angle = (atan2(mousePosition.y - m_Resolution.y / 2,
            mousePosition.x - m_Resolution.x / 2)
            * 180) / 3.141;
        m_Sprite.setRotation(angle);
    }
```

The first portion of the previous code moves the player sprite. The four `if` statements check which of the movement-related Boolean variables (`m_LeftPressed`, `m_RightPressed`, `m_UpPressed`, or `m_DownPressed`) are true and changes `m_Position.x` and `m_Position.y` accordingly. The same formula, from the previous two projects, to calculate the amount to move is also used:

```
position (+ or -) speed * elapsed time.
```

After these four `if` statements, `m_Sprite.setPosition` is called and `m_Position` is passed in. The sprite has now been adjusted by exactly the right amount for that one frame.

The next four `if` statements check whether `m_Position.x` or `m_Position.y` is beyond any of the edges of the current arena. Remember that the confines of the current arena were stored in `m_Arena`, in the `spawn` function. Let's look at the first one of these four `if` statements to understand them all:

```
if (m_Position.x > m_Arena.width - m_TileSize)
{
    m_Position.x = m_Arena.width - m_TileSize;
}
```

The previous code tests to see whether `m_position.x` is greater than `m_Arena.width`, minus the size of a tile (`m_TileSize`). As we will see when we create the background graphics, this calculation will detect the player straying into the wall.

When the `if` statement is `true`, the `m_Arena.width - m_TileSize` calculation is used to initialize `m_Position.x`. This means that the center of the player graphic will never be able to stray past the left-hand edge of the right-hand wall.

The next three `if` statements, which follow the one we have just discussed, do the same thing but for the other three walls.

The last two lines in the preceding code calculate and set the angle that the player sprite is rotated to (that is, facing). This line of code might look a little complex, so let's dig a little deeper.

First, here is the code again for reference:

```
 // Calculate the angle the player is facing
float angle = (atan2(mousePosition.y - m_Resolution.y / 2,
     mousePosition.x - m_Resolution.x / 2)
     * 180) / 3.141;
 m_Sprite.setRotation(angle);
```

In summary, the code calculates the angle between the center of the screen (assumed to be (`m_Resolution.x / 2, m_Resolution.y / 2`)) and the current mouse position. It then sets the rotation of the sprite representing the player based on this angle.

First, the code calculates the angle:

```
atan2(mousePosition.y - m_Resolution.y / 2, mousePosition.x - m_
```

The `atan2` function is used to calculate the angle formed by an imaginary line formed between the center of the screen (`m_Resolution.x / 2, m_Resolution.y / 2`) and the current mouse position (`mousePosition.x, mousePosition.y`).

The result of this calculation is in radians, but SFML works in degrees. The next part of that same line of code converts from radians to degrees:

```
* 180
```

Multiplying by `180` converts it to degrees. Next, dividing by 3.141, which is Pi, makes the angle in the range 0 through 360. This means that the angle is within a full circle.

```
/ 3.141
```

Finally, we set the sprite's rotation:

```
m_Sprite.setRotation(angle);
```

As an aside, I have drastically oversimplified the way the `atan` function works behind the scenes, but that is what functions are for. That's my excuse and I'm sticking to it. If you want to dig deeper into the C++ math library, you can do so.

If you want to explore trigonometric functions in more detail, you can do so here: http://www.cplusplus.com/reference/cmath/.

The last three functions we will add for the `Player` class make the player 20% faster, increase the player's health by 20%, and increase the player's health by the amount passed in, respectively.

Add the following code at the end of the `Player.cpp` file, and then we will take a closer look at it:

```
void Player::upgradeSpeed()
{
    // 20% speed upgrade
    m_Speed += (START_SPEED * .2);
}
void Player::upgradeHealth()
{
    // 20% max health upgrade
    m_MaxHealth += (START_HEALTH * .2);
```

```
}
void Player::increaseHealthLevel(int amount)
{
    m_Health += amount;
    // But not beyond the maximum
if (m_Health > m_MaxHealth)
    {
        m_Health = m_MaxHealth;
    }
}
```

In the preceding code, the `upgradeSpeed()` and `upgradeHealth()` functions increase the value stored in `m_Speed` and `m_MaxHealth`, respectively. These values are increased by 20% by multiplying the starting values by `.2` and adding them to the current values. These functions will be called from the `main` function when the player is choosing what attributes of their character they wish to improve (that is, level up) between levels.

The `increaseHealthLevel()` function takes an `int` value from `main` in the `amount` parameter. This `int` value will be provided by a class called `Pickup`, which we will write in *Chapter 12*, *Collision Detection, Pickups, and Bullets*.

The `m_Health` member variable is increased by the passed-in value. However, there is a catch for the player. The `if` statement checks whether `m_Health` has exceeded `m_MaxHealth` and, if it has, sets it to `m_MaxHealth`. This means the player cannot simply gain infinite health from pick-ups. Instead, they must carefully balance the upgrades they choose between levels.

Of course, our `Player` class can't do anything until we instantiate it and put it to work in our game loop. Before we do that, let's look at the concept of a game camera.

Controlling the game camera with SFML View

In my opinion, the SFML `View` class is one of the most useful classes. After finishing this book, if you make games without using a media/gaming library, you will really notice the absence of `View`.

The `View` class allows us to consider our game as taking place in its own world, with its own properties. What do I mean? Well, when we create a game, we are usually trying to create a virtual world. That virtual world rarely, if ever, is measured in pixels, and rarely, if ever, will that world be the same number of pixels as the player's monitor. We need a way to abstract the virtual world we are building so that it can be of whatever size or shape we like.

Another way to think of SFML `View` is as a camera through which the player views a part of our virtual world. Most games will have more than one camera/view of the world.

For example, consider a split-screen game where two players can be in different parts of the world at the same time. Or, consider a game where there is a small area of the screen that represents the entire game world, but at a very high level/zoomed out, like a minimap.

Even if our games are much simpler than the previous two examples and don't need split screens or minimaps, we will likely want to create a world that is bigger than the screen it is being played on. This is, of course, the case with Zombie Arena.

Additionally, if we are constantly moving the game camera around to show different parts of the virtual world (usually to track the player), what happens to the HUD? If we draw the score and other on-screen HUD

information and then scroll the world around to follow the player, the score will move relative to that camera.

The SFML `View` class easily enables all of these features and solves this problem with very straightforward code. The trick is to create an instance of `View` for every camera – perhaps a `View` instance for the minimap, a `View` instance for the scrolling game world, and then a `View` instance for the HUD.

The instances of `View` can be moved around, sized, and positioned as required. So, the main `View` instance following the game can track the player, the minimap view can remain in a fixed, zoomed-out small corner of the screen, and the HUD can overlay the entire screen and never move, despite the fact that the main `View` instance will go wherever the player goes.

Let's look at some code using a few instances of `View`.

This code is being used to introduce the `View` class. Don't add this code to the Zombie Arena project.

Create and initialize a few instances of `View`:

```
// Create a view to fill a 1920 x 1080 monitor
View mainView(sf::FloatRect(0, 0, 1920, 1080));
// Create a view for the HUD
View hudView(sf::FloatRect(0, 0, 1920, 1080));
```

The previous code creates two `View` objects that fill a 1920 x 1080 monitor. Now, we can do some magic with `mainView` while leaving `hudView` completely alone:

```
// In the update part of the game
// There are lots of things you can do with a View
```

```
// Make the view centre around the player
mainView.setCenter(player.getCenter());
// Rotate the view 45 degrees
mainView.rotate(45)
// Note that hudView is totally unaffected by the previous code
```

When we manipulate the properties of a `View` instance, we do so like this. When we draw sprites, text, or other objects to a view, we must specifically **set** the view as the current view for the window:

```
// Set the current view
window.setView(mainView);
```

Now, we can draw everything we want into that view:

```
// Do all the drawing for this view
window.draw(playerSprite);
window.draw(otherGameObject);
// etc
```

The player might be at any coordinate whatsoever; it doesn't matter because `mainView` is centered around the graphic representing the player.

Now, we can draw the HUD into `hudView`. Note that just like we draw individual elements (background, game objects, text, and so on) in layers from back to front, we also draw views from back to front as well. Hence, a HUD is drawn after the main game scene:

```
// Switch to the hudView
window.setView(hudView);
// Do all the drawing for the HUD
window.draw(scoreText);
```

```
window.draw(healthBar);
// etc
```

Finally, we can draw/show the window and all its views for the current frame in the usual way:

```
window.display();
```

If you want to take your understanding of SFML `View` further than is necessary for this project, including how to achieve split screens and minimaps, then the best guide on the web is on the official SFML website: https://www.sfml-dev.org/tutorials/2.5/graphics-view.php.

Now that we have learned about `View`, we can start coding the Zombie Arena `main` function and use our first `View` instance for real. In *Chapter 13, Layering Views and Implementing the HUD*, we will introduce a second instance of `View` for the HUD and layer it over the top of the main `View` instance.

Starting the Zombie Arena game engine

In this game, we will need a slightly upgraded game engine in `main`. We will have an enumeration called `state`, which will track the current state of the game. Then, throughout `main`, we can wrap parts of our code so that different things happen in different states.

When we created the project, Visual Studio created a file for us called `ZombieArena.cpp`. This will be the file that contains our `main` function and the code that instantiates and controls all our classes.

We begin with the now-familiar `main` function and some `include` directives. Note the addition of an `include` directive for the `Player` class.

Delete the code that Visual Studio added to `ZombieArena.cpp` and add the following code to the `ZombieArena.cpp` file:

```
#include <SFML/Graphics.hpp>
#include "Player.h"
using namespace sf;
int main()
{
    return 0;
}
```

The previous code has nothing new in it except that the `#include "Player.h"` line means we can now use the `Player` class within our code.

Let's flesh out some more of our game engine. The following code does quite a lot. Be sure to read the comments when you add the code to get an idea of what is going on. We will then go through it in more detail.

Add the following highlighted code at the start of the `main` function:

```
int main()
{
 // The game will always be in one of four states
 enum class State { PAUSED, LEVELING_UP,
            GAME_OVER, PLAYING };

 // Start with the GAME_OVER state
    State state = State::GAME_OVER;
 // Get the screen resolution and
```

```
 // create an SFML window
    Vector2f resolution;
    resolution.x =
        VideoMode::getDesktopMode().width;
    resolution.y =
        VideoMode::getDesktopMode().height;
 RenderWindow window(
        VideoMode(resolution.x, resolution.y),
 "Zombie Arena", Style::Fullscreen);
 // Create a an SFML View for the main action
 View mainView(sf::FloatRect(0, 0,
           resolution.x, resolution.y));
 // Here is our clock for timing everything
    Clock clock;
 // How long has the PLAYING state been active
    Time gameTimeTotal;
 // Where is the mouse in
 // relation to world coordinates
    Vector2f mouseWorldPosition;
 // Where is the mouse in
 // relation to screen coordinates
    Vector2i mouseScreenPosition;
 // Create an instance of the Player class
    Player player;
 // The boundaries of the arena
    IntRect arena;
 // The main game loop
 while (window.isOpen())
    {

    }
return 0;
}
```

Let's run through each section of all the code that we entered. Just inside the `main` function, we have the following code:

```
// The game will always be in one of four states
enum class State { PAUSED, LEVELING_UP, GAME_OVER, PLAYING };
```

```
// Start with the GAME_OVER state
State state = State::GAME_OVER;
```

The previous code creates a new enumeration class called `State`. Then, the code creates an instance of the `State` class called `state`. The `state` enumeration can now be one of four values, as defined in the declaration. Those values are `PAUSED`, `LEVELING_UP`, `GAME_OVER`, and `PLAYING`. These four values will be just what we need for keeping track of and responding to the different states that the game can be in at any given time. Note that it is not possible for `state` to hold more than one value at a time.

Immediately after, we added the following code:

```
// Get the screen resolution and create an SFML window
Vector2f resolution;
resolution.x = VideoMode::getDesktopMode().width;
resolution.y = VideoMode::getDesktopMode().height;
RenderWindow window(VideoMode(resolution.x, resolution.y),
 "Zombie Arena", Style::Fullscreen);
```

The previous code declares a `Vector2f` instance called `resolution`. We initialize the two member variables of `resolution` (`x` and `y`) by calling the `VideoMode::getDesktopMode` function for both `width` and `height`. The `resolution` object now holds the resolution of the monitor on which the game is running. The final line of code creates a new `RenderWindow` instance called `window` using the appropriate resolution.

The following code creates an SFML `View` object. The view is positioned (initially) at the exact coordinates of the pixels of the monitor. If we were to use this `View` to do some drawing in this current position, it would be the same as drawing a window without a view. However, we will eventually start to move this view to focus on the parts of our game world that the

player needs to see. Then, when we start to use a second `View` instance, which remains fixed (for the HUD), we will see how this `View` instance can track the action while the other remains static to display the HUD:

```
// Create a an SFML View for the main action
View mainView(sf::FloatRect(0, 0, resolution.x, resolution.y));
```

Next, we created a `Clock` instance to do our timing and a `Time` object called `gameTimeTotal` that will keep a running total of the game time that has elapsed. As the project progresses, we will also introduce more variables and objects to handle timing:

```
// Here is our clock for timing everything
Clock clock;
// How long has the PLAYING state been active
Time gameTimeTotal;
```

The following code declares two vectors: one holding two `float` variables, called `mouseWorldPosition`, and one holding two integers, called `mouseScreenPosition`. The mouse pointer is something of an anomaly because it exists in two different coordinate spaces. We could think of these as parallel universes if we like. Firstly, as the player moves around the world, we will need to keep track of where the crosshair is in that world.

These will be floating-point coordinates and will be stored in `mouseWorldCoordinates`. Of course, the actual pixel coordinates of the monitor itself never change. They will always be 0,0 to horizontal resolution -1 and vertical resolution -1. We will track the mouse pointer position that is relative to this coordinate space using the integers stored in `mouseScreenPosition`:

```
// Where is the mouse in relation to world coordinates
Vector2f mouseWorldPosition;
// Where is the mouse in relation to screen coordinates
Vector2i mouseScreenPosition;
```

Finally, we get to use our `Player` class. This line of code will cause the constructor function (`Player::Player`) to execute. Refer to `Player.cpp` if you want to refresh your memory about this function:

```
// Create an instance of the Player class
Player player;
```

This `IntRect` object will hold starting horizontal and vertical coordinates, as well as a width and a height. Once initialized, we will be able to access the size and location details of the current arena with code such as `arena.left`, `arena.top`, `arena.width`, and `arena.height`:

```
// The boundaries of the arena
IntRect arena;
```

The last part of the code that we added previously is, of course, our game loop:

```
// The main game loop
while (window.isOpen())
{
}
```

You have probably noticed that the code is getting quite long. We'll talk about this inconvenience in the following section.

Managing the code files

One of the advantages of abstraction using classes and functions is that the length (number of lines) of our code files can be reduced. Even though we will be using more than a dozen code files for this project, the length of the code in `ZombieArena.cpp` will still get a little unwieldy toward the end. In the next and final project, we will look at even more ways to abstract and manage our code.

For now, use this tip to keep things more manageable. Notice that on the left-hand side of the code editor in Visual Studio, there are several + and - signs, one of which is shown in this diagram:

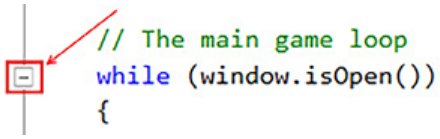


Figure 8.3: Signs on Code editor in Visual Studio

There will be one sign for each block (`if`, `while`, `for`, and so on) of the code. You can expand and collapse these blocks by clicking on the + and - signs. I recommend keeping all the code not currently under discussion collapsed. This will make things much clearer.

Furthermore, we can create our own collapsible blocks. I suggest making a collapsible block out of all the code before the start of the main game loop. To do so, highlight the code and then *right-click* and choose **Outlining** | **Hide Selection**, as shown in the following screenshot:

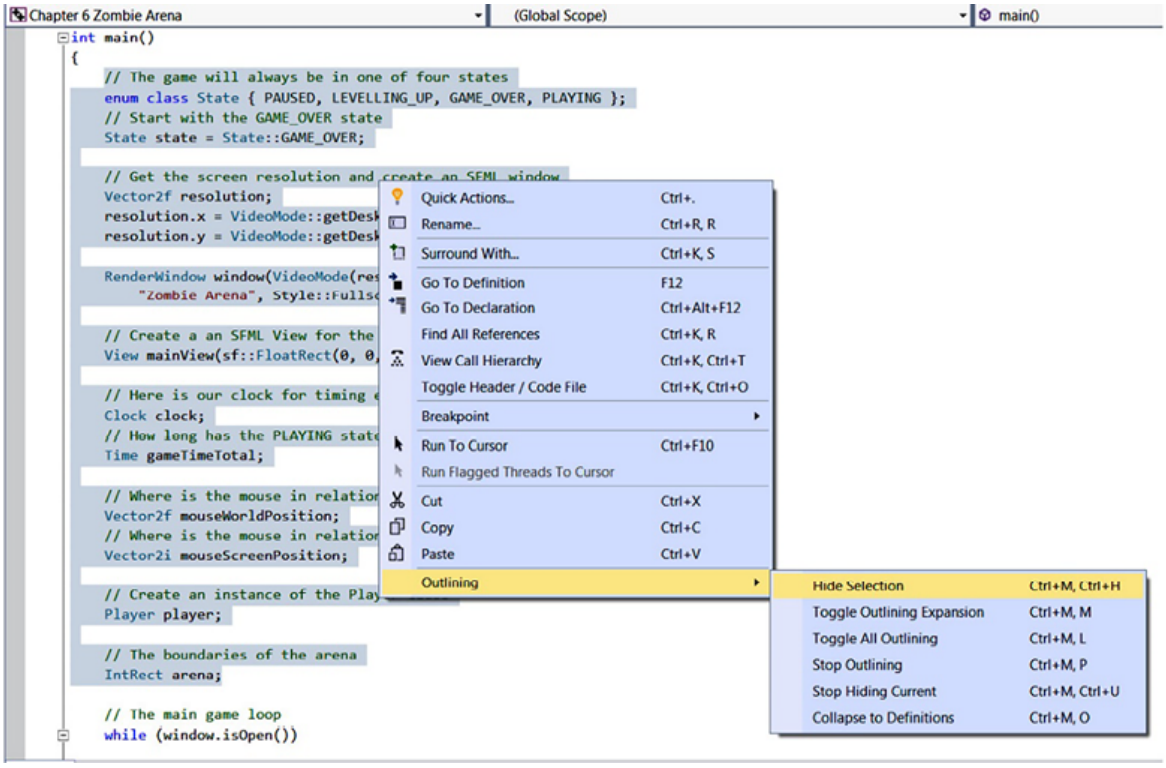

Chapter 6 Zombie Arena
(Global Scope)
main()
int main()
{
// The game will always be in one of four states
enum class State { PAUSED, LEVELLING_UP, GAME_OVER, PLAYING };
// Start with the GAME_OVER state
State state = State::GAME_OVER;
// Get the screen resolution and create an SFML window
Vector2f resolution;
resolution.x = VideoMode::getDesk
resolution.y = VideoMode::getDesk
RenderWindow window(VideoMode(res
"Zombie Arena", Style::Fullsc
// Create a an SFML View for the
View mainView(sf::FloatRect(0, 0,
// Here is our clock for timing
Clock clock;
// How long has the PLAYING state
Time gameTimeTotal;
// Where is the mouse in relation
Vector2f mouseWorldPosition;
// Where is the mouse in relation
Vector2i mouseScreenPosition;
// Create an instance of the Play
Player player;
// The boundaries of the arena
IntRect arena;
// The main game loop
while (window.isOpen())
Quick Actions... Ctrl+.
Rename... Ctrl+R, R
Surround With... Ctrl+K, S
Go To Definition F12
Go To Declaration Ctrl+Alt+F12
Find All References Ctrl+K, R
View Call Hierarchy Ctrl+K, Ctrl+T
Toggle Header / Code File Ctrl+K, Ctrl+O
Breakpoint
Run To Cursor Ctrl+F10
Run Flagged Threads To Cursor
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Outlining
Hide Selection Ctrl+M, Ctrl+H
Toggle Outlining Expansion Ctrl+M, M
Toggle All Outlining Ctrl+M, L
Stop Outlining Ctrl+M, P
Stop Hiding Current Ctrl+M, Ctrl+U
Collapse to Definitions Ctrl+M, O

Figure 8.4: Making a collapsible block

Now, you can click the - and + signs to expand and collapse the block. Each time we add code before the main game loop (and that will be quite often), you can expand the code, add the new lines, and then collapse it again. The following screenshot shows what the code looks like when it is collapsed:

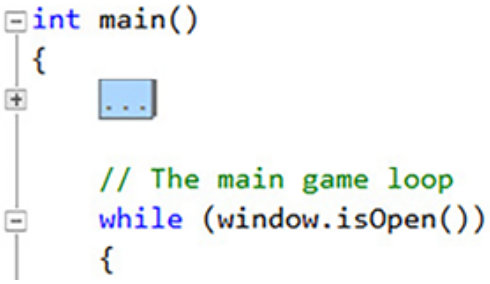


Figure 8.5: A collapsed code

This is much more manageable than it was before. Now, we can make a start with the main game loop.

Starting to code the main game loop

As you can see, the last part of the preceding code is the game loop (`while (window.isOpen()){}`). We will turn our attention to this now. Specifically, we will be coding the input handling section of the game loop.

The code that we will be adding is quite long, but there is nothing complicated about it, though, and we will examine it all in a moment.

Add the following highlighted code to the game loop:

```
// The main game loop
while (window.isOpen())
{
 /*
    ************
    Handle input
    ************
    */
 // Handle events by polling
    Event event;
 while (window.pollEvent(event))
    {
 if (event.type == Event::KeyPressed)
        {
 // Pause a game while playing
 if (event.key.code == Keyboard::Return &&
                state == State::PLAYING)
            {
                state = State::PAUSED;
            }
 // Restart while paused
 else if (event.key.code == Keyboard::Return &&
                state == State::PAUSED)
            {
                state = State::PLAYING;
 // Reset the clock so there isn't a frame jump
```

```
                clock.restart();
            }
        // Start a new game while in GAME_OVER state
        else if (event.key.code == Keyboard::Return &&
                state == State::GAME_OVER)
            {
                state = State::LEVELING_UP;
            }
        if (state == State::PLAYING)
            {
            }
        }
    }// End event polling
}// End game loop
```

In the preceding code, we instantiate an object of the `Event` type. We will use `event`, like we did in the previous projects, to poll for system events. To do so, we wrap the rest of the code from the previous block in a `while` loop with the `window.pollEvent(event)` condition. This will keep looping each frame until there are no more events to process.

Inside this `while` loop, we handle the events we are interested in. First, we test for `Event::KeyPressed` events. If the *Return* key is pressed while the game is in the `PLAYING` state, then we switch `state` to `PAUSED`.

If the *Return* key is pressed while the game is in the `PAUSED` state, then we switch `state` to `PLAYING` and restart the `clock` object. The reason we restart `clock` after switching from `PAUSED` to `PLAYING` is because, while the game is paused, the elapsed time still accumulates. If we didn't restart the clock, all our objects would update their locations as if the frame had just taken a very long time. This will become more apparent as we flesh out the rest of the code in this file.

We then have an `else if` block to test whether the *Return* key was pressed while the game was in the `GAME_OVER` state. If it was, then `state` is changed

to `LEVELING_UP`.

Note that the `GAME_OVER` state is the state where the home screen is displayed. So, the `GAME_OVER` state is the state after the player has just died and when the player first runs the game. The first thing that the player gets to do each game is pick an attribute to improve (that is, level up).

In the previous code, there is a final `if` condition to test whether the state is equal to `PLAYING`. This `if` block is empty and we will add code to it throughout the project.

We will add code to lots of different parts of this file throughout the project. Therefore, it is worthwhile taking the time to understand the different states our game can be in and where we handle them. It will also be very beneficial to collapse and expand the different `if`, `else`, and `while` blocks as and when appropriate.

Spend some time thoroughly familiarizing yourself with the `while`, `if`, and `else if` blocks we have just coded. We will be referring to them regularly.

Next, immediately after the previous code and still inside the game loop, which is still dealing with handling input, add the following highlighted code. Note the existing code (not highlighted) that shows exactly where the new (highlighted) code goes:

```
        }// End event polling
    // Handle the player quitting
    if (Keyboard::isKeyPressed(Keyboard::Escape))
```

```
    {
        window.close();
    }
 // Handle WASD while playing
 if (state == State::PLAYING)
    {
 // Handle the pressing and releasing of WASD keys
 if (Keyboard::isKeyPressed(Keyboard::W))
        {
            player.moveUp();
        }
 else
        {
            player.stopUp();
        }
 if (Keyboard::isKeyPressed(Keyboard::S))
        {
            player.moveDown();
        }
 else
        {
            player.stopDown();
        }
 if (Keyboard::isKeyPressed(Keyboard::A))
        {
            player.moveLeft();
        }
 else
        {
            player.stopLeft();
        }
 if (Keyboard::isKeyPressed(Keyboard::D))
        {
            player.moveRight();
        }
 else
        {
            player.stopRight();
        }
    }// End WASD while playing
}// End game loop
```

In the preceding code, we first test to see whether the player has pressed the *Escape* key. If it is pressed, the game window will be closed.

Next, within one big `if(state == State::PLAYING)` block, we check each of the *WASD* keys in turn. If a key is pressed, we call the appropriate `player.move...` function. If it is not, we call the related `player.stop...` function.

This code ensures that, in each frame, the player object will be updated with the *WASD* keys that are pressed and those that are not. The `player.move...` and `player.stop...` functions store the information in the member Boolean variables (`m_LeftPressed`, `m_RightPressed`, `m_UpPressed`, and `m_DownPressed`). The `Player` class then responds to the value of these Booleans, in each frame, in the `player.update` function, which we will call in the update section of the game loop.

Now, we can handle the keyboard input to allow the player to level up at the start of each game and in between each wave. Add and study the following highlighted code and then we will discuss it:

```
    }// End WASD while playing
 // Handle the LEVELING up state
 if (state == State::LEVELING_UP)
    {
 // Handle the player LEVELING up
 if (event.key.code == Keyboard::Num1)
        {
            state = State::PLAYING;
        }
 if (event.key.code == Keyboard::Num2)
        {
            state = State::PLAYING;
        }
 if (event.key.code == Keyboard::Num3)
        {
            state = State::PLAYING;
```

```
            }
        if (event.key.code == Keyboard::Num4)
            {
                state = State::PLAYING;
            }
        if (event.key.code == Keyboard::Num5)
            {
                state = State::PLAYING;
            }
        if (event.key.code == Keyboard::Num6)
            {
                state = State::PLAYING;
            }

        if (state == State::PLAYING)
            {
        // Prepare the level
        // We will modify the next two lines later
                arena.width = 500;
                arena.height = 500;
                arena.left = 0;
                arena.top = 0;
        // We will modify this line of code later
        int tileSize = 50;
        // Spawn the player in middle of the arena
                player.spawn(arena, resolution, tileSize);

        // Reset clock so there isn't a frame jump
                clock.restart();
            }
        }// End LEVELING up

    }// End game loop
```

In the preceding code, which is all wrapped in a test to see whether the current value of `state` is equal to `LEVELING_UP`, we handle the keyboard keys *1*, *2*, *3*, *4*, *5*, and *6*. In the `if` block for each, we simply set `state` to `State::PLAYING`. We will add some code to deal with each level-up option later in *Chapter 14*, *Sound Effects, File I/O, and Finishing the Game*.

This code does the following things:

1. If the `state` is equal to `LEVELING_UP`, wait for either the *1*, *2*, *3*, *4*, *5*, or *6* key to be pressed.
2. When pressed, change `state` to `PLAYING`.
3. When the `state` changes, still within the `if (state == State::LEVELING_UP)` block, the nested `if(state == State::PLAYING)` block will run.
4. Within this block, we set the location and size of `arena`, set the `tileSize` to `50`, pass all the information to `player.spawn`, and call `clock.restart`.

Now, we have an actual spawned player object that is aware of its environment and can respond to key presses. We can now update the scene on each pass through the loop.

Be sure to neatly collapse the code from the input-handling part of the game loop since we are done with that for now. The following code is in the updating part of the game loop. Add and study the following highlighted code and then we can discuss it:

```
    }// End LEVELING up
 /*
    ****************
    UPDATE THE FRAME
    ****************
    */
 if (state == State::PLAYING)
    {
 // Update the delta time
        Time dt = clock.restart();

 // Update the total game time
        gameTimeTotal += dt;

```

```
 // Make a fraction of 1 from the delta time
 float dtAsSeconds = dt.asSeconds();
 // Where is the mouse pointer
        mouseScreenPosition = Mouse::getPosition();
 // Convert mouse position to world
 // based coordinates of mainView
        mouseWorldPosition = window.mapPixelToCoords(
            Mouse::getPosition(), mainView);
 // Update the player
        player.update(dtAsSeconds, Mouse::getPosition());
 // Make a note of the players new position
 Vector2f playerPosition(player.getCenter());

 // Make the view centre
 // the around player
        mainView.setCenter(player.getCenter());
    }// End updating the scene

}// End game loop
```

Note that the previous code is wrapped in a test to make sure the game is in the `PLAYING` state. We don't want this code to run if the game has been paused, it has ended, or if the player is choosing what to level up.

First, we restart the clock and store the time that the previous frame took in the `dt` variable:

```
// Update the delta time
Time dt = clock.restart();
```

Next, we add the time that the previous frame took to the accumulated time the game has been running for, as held by `gameTimeTotal`:

```
// Update the total game time
gameTimeTotal += dt;
```

Now, we initialize a `float` variable called `dtAsSeconds` with the value returned by the `dt.AsSeconds` function. For most frames, this will be a fraction of one. This is perfect for passing to the `player.update` function to be used to calculate how much to move the player's sprite.

Now, we can initialize `mouseScreenPosition` using the `MOUSE::getPosition` function.

You are probably wondering about the slightly unusual syntax for getting the position of the mouse. This is called a **static function**. If we define a function in a class with the `static` keyword, we can call that function using the class name and without an instance of the class. C++ OOP has lots of quirks and rules like this. We will see more as we progress.

We then initialize `mouseWorldPosition` using the SFML `mapPixelToCoords` function on the `window`. We discussed this function when talking about the `View` class earlier in this chapter.

At this point, we are now able to call `player.update` and pass in `dtAsSeconds` and the position of the mouse, as required.

We store the player's new center in a `Vector2f` instance called `playerPosition`. At the moment, this is unused, but we will have a use for this later in the project.

We can then center the view around the center of the player's up-to-date position with `mainView.setCenter(player.getCenter()).`

We are now able to draw the player to the screen. Add the following highlighted code, which splits the draw section of the main game loop into different states:

```
        }// End updating the scene
 /*
        **************
        Draw the scene
        **************
        */
 if (state == State::PLAYING)
        {
            window.clear();
 // set the mainView to be displayed in the window
 // And draw everything related to it
            window.setView(mainView);
 // Draw the player
            window.draw(player.getSprite());
        }
 if (state == State::LEVELING_UP)
        {
        }
 if (state == State::PAUSED)
        {
        }
 if (state == State::GAME_OVER)
        {
        }
        window.display();
    }// End game loop
return 0;
}
```

Within the `if(state == State::PLAYING)` section of the previous code, we clear the screen, set the view of the window to `mainView`, and then draw the player sprite with `window.draw(player.getSprite())`.

After all the different states have been handled, the code shows the scene in the usual manner with `window.display();`.

You can run the game and see our player character spin around in response to moving the mouse.

When you run the game, you need to press *Enter* to start the game, and then select a number from *1* to *6* to simulate choosing an upgrade option. Then, the game will start.

You can also move the player around within the (empty) 500 x 500 pixel arena. You can see our lonely player in the center of the screen, as shown here:

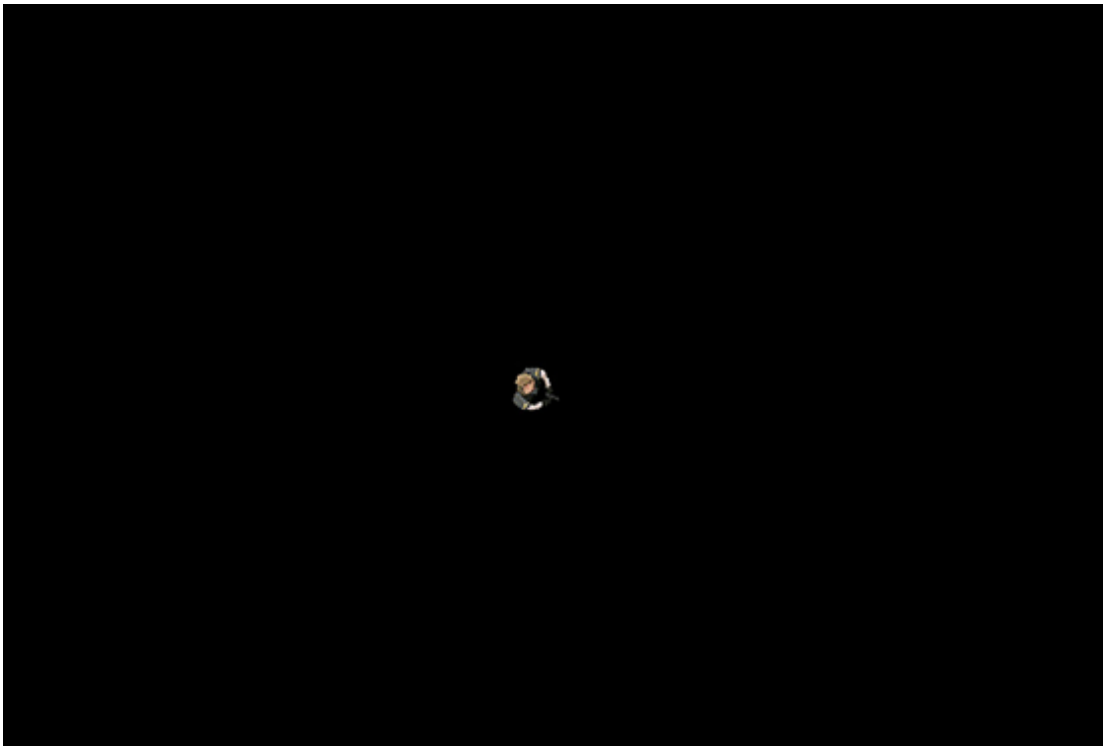


Figure 8.6: Lonely player in the center of the screen

You can't, however, get any sense of movement because we haven't implemented the background. We will do so in the next chapter.

Summary

Phew! That was a long one. We have done a lot in this chapter: we built our first class for the Zombie Arena project, `Player`, and put it to use in the game loop. We also learned about and used an instance of the `View` class, although we haven't explored the benefits this gives us just yet.

In the next chapter, we will build our arena background by exploring what sprite sheets are. We will also learn about C++ **references**, which allow us to manipulate variables, even when they are out of scope. Out of scope means the variables are in another function.

Frequently asked questions

Q) I noticed something odd in the code we have been writing. In `if` statements, such as the following:

```
if (event.type == Event::KeyPressed)…
```

How does the `Event` parameter passed into the `pollEvent` function end up being used? After all, don't the variables and objects only have scope in the function in which they are declared?

A) The reason is C++ references. References in C++ are variables that act as aliases for other variables. In the code under discussion, there are no explicit references. However, references are used to pass objects to functions efficiently, avoiding unnecessary copying. As the parameter of the `pollEvent` function is defined as a reference, values can be assigned to the

passed-in event object and those values persist in our `main` function. We will understand this more when we discuss references in the next chapter.

Q) I noticed we have coded quite a few functions of the `Player` class that we don't use. Why is this?

A) Rather than coming back to the `Player` class, we have added all the code that we will need throughout the project. By the end of *Chapter 14*, *Sound Effects, File I/O, and Finishing the Game*, we will have made full use of all of these functions.

9

C++ References, Sprite Sheets, and Vertex Arrays

In *Chapter 4*, *Loops, Arrays, Switch, Enumerations, and Functions – Implementing Game Mechanics*, we talked about **scope**. This is the concept that variables declared in a function or inner block of code only have scope (that is, can be seen or used) in that function or block. Using only the C++ knowledge we have currently can cause a problem. What do we do if we need to work on a few complex objects that are needed in the `main` function? This could imply that all the code must be in the `main` function.

In this chapter, we will explore C++ **references**, which allow us to work on variables and objects that are otherwise out of scope. In addition to this, these references will help us avoid having to pass large objects between functions, which is a slow process. It is slow because each time we do this, a copy of the variable or object must be made.

Armed with this new knowledge of references, we will look at the SFML `VertexArray` class, which allows us to build up a large image that can be quickly and efficiently drawn to the screen using multiple parts in a single image file. By the end of this chapter, we will have a scalable, random, scrolling background that's been made using references and a `VertexArray` object.

In this chapter, we will discuss the following topics:

- Understanding C++ references
- SFML vertex arrays and sprite sheets
- Creating a randomly generated scrolling background
- Using the background

Understanding C++ references

When we pass values to a function or return values from a function, that is exactly what we are doing – passing/returning by **value**. What happens is that a copy of the value held by the variable is made and then sent to the function, where it is used.

The significance of this is twofold:

1. If we want the function to make a permanent change to a variable, this system is no good to us.
2. When a copy is made to pass in as an argument or returned from the function, processing power and memory are consumed. For a simple `int`, or even perhaps a `Sprite`, this is insignificant. However, for a complex object, perhaps an entire game world (or background), the copying process will seriously affect our game's performance.

References are the solution to these two problems. A **reference** is a special type of variable. A reference *refers* to another variable. Here is an example to help you understand this better:

```
int numZombies = 100;
int& rNumZombies = numZombies;
```

In the preceding code, we declare and initialize a regular `int` called `numZombies`. We then declare and initialize an `int` reference called `rNumZombies`. The reference operator, `&`, which follows the type, determines that a reference is being declared.

The `r` prefix at the front of the reference name is optional but is useful for remembering that we are dealing with a reference.

Now, we have an `int` variable called `numZombies`, which stores the `100` value, and an `int` reference called `rNumZombies`, which refers to `numZombies`.

Anything we do to `numZombies` can be seen through `rNumZombies`, and anything we do to `rNumZombies`, we are actually doing to `numZombies`. Take a look at the following code:

```
int score = 10;
int& rScore = score;
score ++;
rScore ++;
```

In the previous code, we declare an `int` called `score`. Next, we declare an `int` reference called `rScore` that refers to `score`. Remember that anything we do to `score` can be seen by `rScore` and anything we do to `rScore` is being done to `score`.

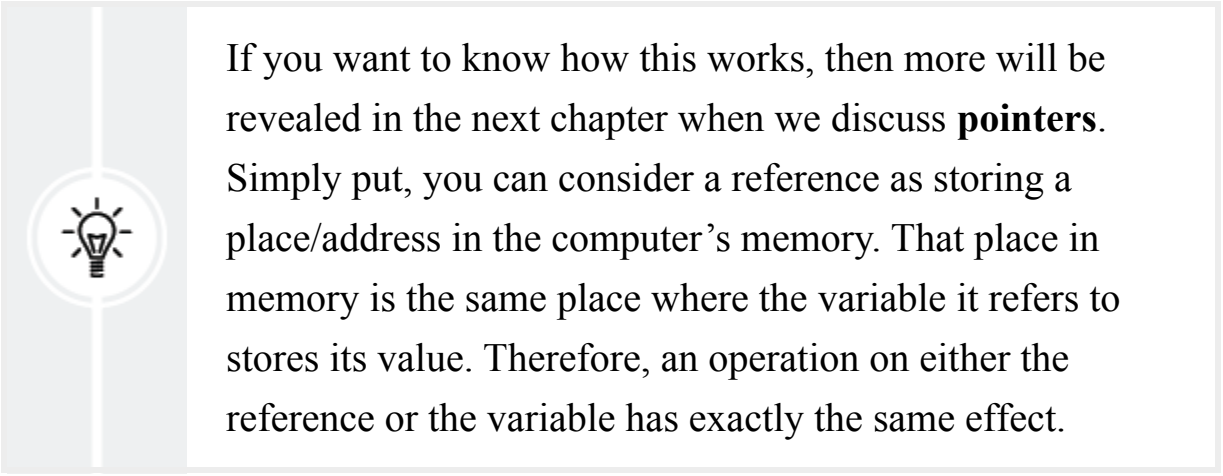
Therefore, consider what happens when we increment `score` like this:

```
score ++;
```

The `score` variable now stores the `11` value. In addition to this, if we were to output `rScore`, it would also output `11`. The next line of code is as follows:

```
rScore ++;
```

Now, `score` actually holds the `12` value because anything we do to `rScore` is done to `score`.



If you want to know how this works, then more will be revealed in the next chapter when we discuss **pointers**. Simply put, you can consider a reference as storing a place/address in the computer's memory. That place in memory is the same place where the variable it refers to stores its value. Therefore, an operation on either the reference or the variable has exactly the same effect.

For now, it is much more important to talk about the *why* of references. There are two reasons to use references, and we have already mentioned them. Here they are, summarized again:

1. Changing/reading the value of a variable/object in another function, which is otherwise out of scope.
2. Passing/returning to/from a function without making a copy (and, therefore, more efficiently).

Study the following code and then we will discuss it:

```
void add(int n1, int n2, int a);
void referenceAdd(int n1, int n2, int& a);
```

```
int main()
{
    int number1 = 2;
    int number2 = 2;
    int answer = 0;

    add(number1, number2, answer);
    // answer equals zero because it is passed as a copy
// Nothing happens to answer in the scope of main
referenceAdd(number1, number2, answer);
    // Now answer is 4 because it was passed by reference
// When the referenceAdd function did this:
// answer = num1 + num 2;
// It is actually changing the value stored by answer
return 0;
}
// Here are the two function definitions
// They are exactly the same except that
// the second passes a reference to a
void add(int n1, int n2, int a)
{
    a = n1 + n2;
    // a now equals 4
// But when the function returns a is lost forever
}
void referenceAdd(int n1, int n2, int& a)
{
    a = n1 + n2;
    // a now equals 4
// But a is a reference!
// So, it is answer, back in main, that equals 4
}
```

The previous code begins with the prototypes of two functions: `add` and `referenceAdd`. The `add` function takes three `int` variables, while the `referenceAdd` function takes two `int` variables and an `int` reference.

When the `add` function is called and the `number1`, `number2`, and `answer` variables are passed in, a copy of the values is made and new variables local

to `add` (that is, `n1`, `n2`, and `a`) are manipulated. As a result of this, the `answer`, back in `main`, remains at zero.

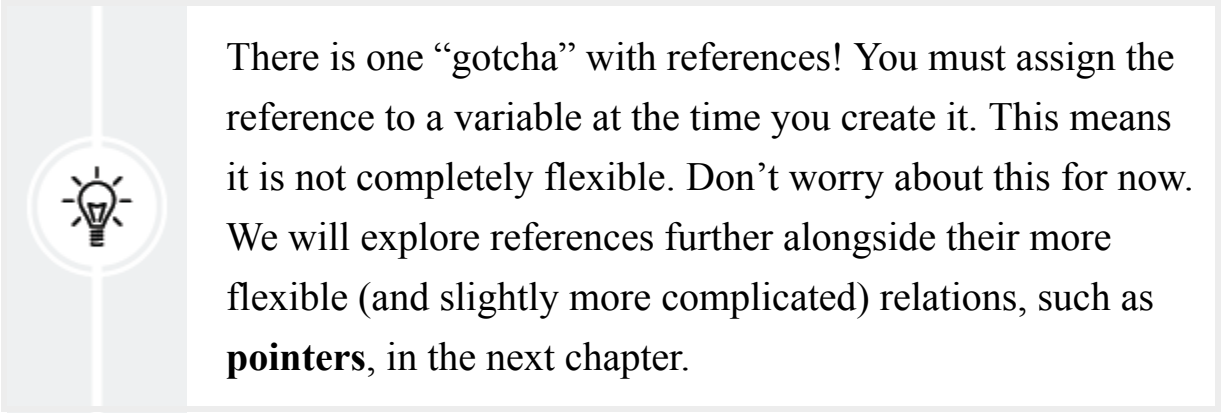
When the `referenceAdd` function is called, `number1` and `number2` are again passed by value. However, `answer` is passed by reference. When the value of `n1` that's added to `n2` is assigned to the reference, `a`, what is really happening is that the value is assigned to `answer` back in the `main` function.

It is probably obvious that we would never need to use a reference for something this simple. It does, however, demonstrate the mechanics of passing by reference.

Now, let's summarize what we know about references.

Summarizing references

The previous code demonstrated how a reference can be used to alter the value of a variable in one scope using code in another. As well as being extremely convenient, passing by reference is also very efficient because no copy is made. Our example, which is using a reference to an `int`, is a bit ambiguous because, as an `int` is so small, there is no real efficiency gain. Later in this chapter, we will use a reference to pass an entire level layout and the efficiency gain will be significant.



There is one "gotcha" with references! You must assign the reference to a variable at the time you create it. This means it is not completely flexible. Don't worry about this for now. We will explore references further alongside their more flexible (and slightly more complicated) relations, such as **pointers**, in the next chapter.

This is largely irrelevant for an `int`, but potentially significant for a large object of a class. We will use this exact technique when we implement the scrolling background of the Zombie Arena game later in this chapter.

Next, we will learn about vertex arrays and sprite sheets.

SFML vertex arrays and sprite sheets

We are nearly ready to implement the scrolling background. We just need to learn about SFML vertex arrays and sprite sheets.

What is a sprite sheet?

A **sprite sheet** is a set of images, either frames of animation or individual graphics, contained in one image file. Take a closer look at this sprite sheet, which contains four separate images that will be used to draw the background in our Zombie Arena game:

Figure 9.1: Sprite sheet

SFML allows us to load a sprite sheet as a regular texture, in the same way we have done for every texture in this book so far. When we load multiple images as a single texture, the GPU can handle it much more efficiently.

> A modern PC could handle these four textures without using a sprite sheet. It is worth learning these techniques, however, as our games are going to start getting progressively more demanding on our hardware.

You could also refer to the sprite sheet as a texture atlas. Typically, the difference between a sprite sheet and a texture atlas is that a sprite sheet usually contains multiple frames for one “thing,” like a character or a background, and the frames are usually packed uniformly – like ours. A texture atlas, on the other hand, usually consists of textures for multiple things, perhaps a whole level or even an entire game, and is likely to be less uniformly arranged and contain textures of different sizes. Furthermore, a texture atlas will often be accompanied by a text file of data describing the names, locations, and sizes of the individual textures. The game would use this text file to access the images it needs. Regardless of what you call the graphics file with multiple images in it, having multiple images in a single file speeds up loading and accessing them during gameplay.

What we need to do when we draw an image from the sprite sheet is make sure we refer to the precise pixel coordinates of the part of the sprite sheet we require, like so:

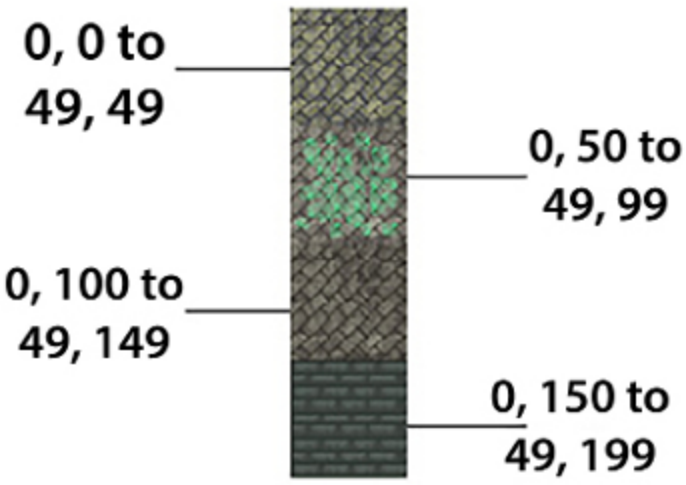


Figure 9.2: Pixel coordinates of the sprite sheet

The previous diagram labels each part/tile with the coordinates and their position within the sprite sheet. These coordinates are called **texture coordinates**. We will use these texture coordinates in our code to draw just the right parts that we require.

What is a vertex array?

First, we need to ask: what is a vertex? A **vertex** is a single graphical point, that is, a coordinate. This point is defined by a horizontal and vertical position. The plural of vertex is vertices. A vertex array is a whole collection of vertices.

In SFML, each vertex in a vertex array also has a color and a related additional vertex (that is, a pair of coordinates) called **texture coordinates**. Texture coordinates are the position of the image we want to use in terms of a sprite sheet. Later, we will see how we can position graphics and choose a part of the sprite sheet to display at each position, all with a single vertex array.

The SFML `VertexArray` class can hold different types of vertex sets. But each `VertexArray` should only hold one type of set. We use the type of set that suits the occasion.

Common scenarios in video games include, but are not limited to, the following **primitive** types:

- **Point**: A single vertex per point.
- **Line**: Two vertices per set that define the start and end point of the line.
- **Triangle**: Three vertices per point. This is the most commonly used (in the thousands) for complex 3D models, or in pairs to create a simple rectangle such as a sprite.
- **Quad**: Four vertices per set. This is a convenient way to map rectangular areas from a sprite sheet.

We will use quads in this project because they are just what we need for rectangular sprites.

Building a background from tiles

The Zombie Arena background will be made up of a random arrangement of square images. You can think of this arrangement like tiles on a floor.

In this project, we will be using vertex arrays with `quad` sets. Each vertex will be part of a set of four (that is, a quad). Each vertex will define one corner of a tile from our background, while each texture coordinate will hold an appropriate value based on a specific image from the sprite sheet.

Let's look at some code to get us started. This isn't the exact code we will use in the project, but it is close enough and allows us to study vertex arrays before we move on to the actual implementation we will use.

Building a vertex array

As we do when we create an instance of a class, we declare our new object. The following code declares a new object of the `VertexArray` type, which we will call `background`:

```
// Create a vertex array
VertexArray background;
```

We want to let our instance of `VertexArray` know which type of primitive we will be using. Remember that points, lines, triangles, and quads all have a different number of vertices. By setting the `VertexArray` instance to hold a specific type, it will be possible to know the start of each primitive. In our case, we want quads. Here is the code that will do this:

```
// What primitive type are we using
background.setPrimitiveType(Quads);
```

As with regular C++ arrays, a `VertexArray` instance needs to be set to a particular size. The `VertexArray` class is more flexible than a regular array. It allows us to change its size while the game is running. The size could be configured at the same time as the declaration, but our background needs to expand with each wave. The `VertexArray` class provides this functionality with the `resize` function. Here is the code that would set the size of our arena to a 10 by 10 tile size:

```
// Set the size of the vertex array
background.resize(10 * 10 * 4);
```

In the previous line of code, the first `10` is the width, the second `10` is the height, and `4` is the number of vertices in a quad. We could have just passed in 400, but showing the calculation like this makes it clear what we are doing. When we code the project for real, we will go a step further to aid clarity and declare variables for each part of the calculation.

We now have a `VertexArray` instance ready to have its hundreds of vertices configured. Here is how we set the position coordinates on the first four vertices (that is, the first quad):

```
// Position each vertex in the current quad
background[0].position = Vector2f(0, 0);
background[1].position = Vector2f(49, 0);
background[2].position = Vector2f(49,49);
background[3].position = Vector2f(0, 49);
```

Here is how we set the texture coordinates of these same vertices to the first image in the sprite sheet.

These coordinates in the image file are from `0,0` (in the top-left corner) to `49,49` (in the bottom-right corner):

```
// Set the texture coordinates of each vertex
background[0].texCoords = Vector2f(0, 0);
background[1].texCoords = Vector2f(49, 0);
background[2].texCoords = Vector2f(49, 49);
background[3].texCoords = Vector2f(0, 49);
```

If we wanted to set the texture coordinates to the second image in the sprite sheet, we would have written the code like this:

```
// Set the texture coordinates of each vertex
background[0].texCoords = Vector2f(0, 50);
```

```
    background[1].texCoords = Vector2f(49, 50);
    background[2].texCoords = Vector2f(49, 99);
    background[3].texCoords = Vector2f(0, 99);
```

Of course, if we define each and every vertex like this individually, then we are going to be spending a long time configuring even a simple 10 by 10 arena.

When we implement our background for real, we will devise a set of nested `for` loops that loop through each quad, pick a random background image, and assign the appropriate texture coordinates.

The code will need to be quite smart. It will need to know when it is an edge tile so that it can use the wall image from the sprite sheet. It will also need to use appropriate variables that know the position of each background tile in the sprite sheet, as well as the overall size of the required arena.

We will make this complexity manageable by putting all the code in both a separate function and a separate file. We will make the `VertexArray` instance usable in `main` by using a C++ reference.

We will examine these details later. You may have noticed that at no point have we associated a texture (the sprite sheet with the vertex array). Let's see how to do that now.

Using the vertex array to draw

Now that we have prepared the vertices and texture coordinates, we are ready to draw to the screen. We can load the sprite sheet as a texture in the same way that we load any other texture, as shown in the following code:

```
// Load the texture for our background vertex array
Texture textureBackground;
textureBackground.loadFromFile("graphics/background_sheet.png")
```

We can then draw the entire `VertexArray` with one call to `draw`:

```
// Draw the background
window.draw(background, &textureBackground);
```

The previous code is much more efficient than drawing every tile as an individual sprite.

Before we move on, notice the slightly odd-looking `&` notation before the `textureBackground` code. Your immediate thought might be that this has something to do with references. What is going on here is we are passing the memory address of the `Texture` instance instead of the actual `Texture` instance. We will learn more about this in the next chapter.

We are now able to use our knowledge of references and vertex arrays to implement the next stage of the Zombie Arena project, which is the randomly generated scrolling background.

Creating a randomly generated scrolling background

In this section, we will create a function that makes a background in a separate file. We will ensure the background will be available (in scope) to the `main` function by using a vertex array reference.

As we will be writing other functions that share data with the `main` function, we will write them all in their own `.cpp` files. We will provide prototypes for these functions in a new header file that we will include (with an `#include` directive) in `ZombieArena.cpp`.

To achieve this, let's make a new header file called `ZombieArena.h`. We are now ready to code the header file for our new function.

In this new `ZombieArena.h` header file, add the following highlighted code, including the function prototype:

```
#pragma once
#include <SFML/Graphics.hpp>
using namespace sf;
int createBackground(VertexArray& rVA, IntRect arena);
```

The previous code allows us to write the definition of a function called `createBackground`. To match the prototype, the function definition must return an `int` value, and receive a `VertexArray` reference and an `IntRect` object as parameters.

Now, we can create a new `.cpp` file in which we will code the function definition. Create a new file called `CreateBackground.cpp`. We are now ready to code the function definition that will create our background.

Add the following code to the `CreateBackground.cpp` file, and then we will review it:

```
#include "ZombieArena.h"
int createBackground(VertexArray& rVA, IntRect arena)
{
    // Anything we do to rVA we are really doing
// to background (in the main function)
// How big is each tile/texture
const int TILE_SIZE = 50;
    const int TILE_TYPES = 3;
    const int VERTS_IN_QUAD = 4;
    int worldWidth = arena.width / TILE_SIZE;
    int worldHeight = arena.height / TILE_SIZE;
    // What type of primitive are we using?
    rVA.setPrimitiveType(Quads);
    // Set the size of the vertex array
    rVA.resize(worldWidth * worldHeight * VERTS_IN_QUAD);
    // Start at the beginning of the vertex array
int currentVertex = 0;
    return TILE_SIZE;
}
```

In the previous code, we write the function signature as well as the opening and closing curly brackets that mark the function body.

Within the function body, we declare and initialize three new `int` constants to hold values that we will need to refer to throughout the rest of the function. They are `TILE_SIZE`, `TILE_TYPES` and `VERTS_IN_QUAD`.

The `TILE_SIZE` constant refers to the size in pixels of each tile within the sprite sheet. The `TILE_TYPES` constant refers to the number of different tiles within the sprite sheet. We could add more tiles to our sprite sheet and change `TILE_TYPES` to match the change, and the code we are about to write would still work. `VERTS_IN_QUAD` refers to the fact that there are four vertices in every quad. It is less error prone to use this constant compared to always typing the number `4`, which is less clear.

We then declare and initialize two `int` variables: `worldWidth` and `worldHeight`. These variables might appear obvious as to their use. They are betrayed by their names, but it is worth pointing out that they refer to the width and height of the world in the number of tiles, not pixels. The `worldWidth` and `worldHeight` variables are initialized by dividing the height and width of the passed-in arena by the `TILE_SIZE` constant.

Next, we get to use our reference for the first time. Remember that anything we do to `rVA`, we are really doing to the variable that was passed in, which is in scope in the `main` function (or will be when we code it).

Then, we prepare the vertex array to use quads using `rVA.setType` and then we make it just the right size by calling `rVA.resize`. To the `resize` function, we pass in the result of `worldWidth * worldHeight * VERTS_IN_QUAD`, which equates to the exact number of vertices that our vertex array will have when we are done preparing it.

The last line of code declares and initializes `currentVertex` to zero. We will use `currentVertex` as we loop through the vertex array, initializing all the vertices.

We can now write the first part of a nested `for` loop that will prepare the vertex array. Add the following highlighted code and, based on what we have learned about vertex arrays, try and work out what it does:

```
 // Start at the beginning of the vertex array
int currentVertex = 0;
 for (int w = 0; w < worldWidth; w++)
    {
 for (int h = 0; h < worldHeight; h++)
        {
 // Position each vertex in the current quad
            rVA[currentVertex + 0].position =
```

```
 Vector2f(w * TILE_SIZE, h * TILE_SIZE);

            rVA[currentVertex + 1].position =
 Vector2f((w * TILE_SIZE) + TILE_SIZE, h * TILE_SIZE);

            rVA[currentVertex + 2].position =
 Vector2f((w * TILE_SIZE) + TILE_SIZE, (h * TILE_SIZE)
                + TILE_SIZE);

            rVA[currentVertex + 3].position =
 Vector2f((w * TILE_SIZE), (h * TILE_SIZE)
                + TILE_SIZE);


 // Position ready for the next four vertices
            currentVertex = currentVertex + VERTS_IN_QUAD;
        }
    }
    return TILE_SIZE;
}
```

The code that we just added steps through the vertex array by using a nested `for` loop, which first steps through the first four vertices: `currentVertex + 1, currentVertex + 2`, and so on.

We access each vertex in the array using the array notation, `rvA[currentVertex + 0]..`, and so on. Using the array notation, we call the `position` function, `rvA[currentVertex + 0].position...`.

To the `position` function, we pass the horizontal and vertical coordinates of each vertex. We can work these coordinates out programmatically by using a combination of `w`, `h`, and `TILE_SIZE`.

At the end of the previous code, we position `currentVertex`, ready for the next pass through the nested `for` loop by advancing it four places (that is,

adding four) with the code, that is, `currentVertex = currentVertex + VERTS_IN_QUAD`.

Of course, all this does is set the coordinates of our vertices; it doesn't assign a texture coordinate from the sprite sheet. This is what we will do next.

To make it absolutely clear where the new code goes, I have shown it in context, along with all the code that we wrote a moment ago. Add and study the following highlighted code:

```
for (int w = 0; w < worldWidth; w++)
    {
        for (int h = 0; h < worldHeight; h++)
        {
            // Position each vertex in the current quad
            rVA[currentVertex + 0].position =
                Vector2f(w * TILE_SIZE, h * TILE_SIZE);

            rVA[currentVertex + 1].position =
                Vector2f((w * TILE_SIZE) + TILE_SIZE, h * TILE_

            rVA[currentVertex + 2].position =
                Vector2f((w * TILE_SIZE) + TILE_SIZE, (h * TILE
                + TILE_SIZE);

            rVA[currentVertex + 3].position =
                Vector2f((w * TILE_SIZE), (h * TILE_SIZE)
                + TILE_SIZE);


 // Define the position in the Texture for current quad
 // Either grass, stone, bush or wall
 if (h == 0 || h == worldHeight-1 ||
                w == 0 || w == worldWidth-1)
            {
 // Use the wall texture
                rVA[currentVertex + 0].texCoords =
 Vector2f(0, 0 + TILE_TYPES * TILE_SIZE);
```

```
                rVA[currentVertex + 1].texCoords =
 Vector2f(TILE_SIZE, 0 +
                    TILE_TYPES * TILE_SIZE);

                rVA[currentVertex + 2].texCoords =
 Vector2f(TILE_SIZE, TILE_SIZE +
                    TILE_TYPES * TILE_SIZE);

                rVA[currentVertex + 3].texCoords =
 Vector2f(0, TILE_SIZE +
                    TILE_TYPES * TILE_SIZE);
            }
// Position ready for the next for vertices
            currentVertex = currentVertex + VERTS_IN_QUAD;
        }
    }
    return TILE_SIZE;
}
```

The preceding highlighted code sets up the coordinates within the sprite sheet that each vertex is related to. Notice the somewhat long `if` condition. The condition checks whether the current quad is either one of the very first or the very last quads in the arena. If it is (one of the first or last), then this means it is part of the boundary. We can then use a simple formula using `TILE_SIZE` and `TILE_TYPES` to target the wall texture from the sprite sheet.

The array notation and the `texCoords` member are initialized for each vertex, in turn, to assign the appropriate corner of the wall texture within the sprite sheet.

The following code is wrapped in an `else` block. This means that it will run through the nested `for` loop each time the quad does not represent a border/wall tile. Add the following highlighted code among the existing code, and then we will examine it:

```
 // Define position in Texture for current quad
// Either grass, stone, bush or wall
if (h == 0 || h == worldHeight-1 ||
                w == 0 || w == worldWidth-1)
            {
                // Use the wall texture
                rVA[currentVertex + 0].texCoords =
                    Vector2f(0, 0 + TILE_TYPES * TILE_SIZE);

                rVA[currentVertex + 1].texCoords =
                    Vector2f(TILE_SIZE, 0 +
                    TILE_TYPES * TILE_SIZE);

                rVA[currentVertex + 2].texCoords =
                    Vector2f(TILE_SIZE, TILE_SIZE +
                    TILE_TYPES * TILE_SIZE);

                rVA[currentVertex + 3].texCoords =
                    Vector2f(0, TILE_SIZE +
                    TILE_TYPES * TILE_SIZE);
            }
 else
            {
 // Use a random floor texture
 srand((int)time(0) + h * w - h);
 int mOrG = (rand() % TILE_TYPES);
 int verticalOffset = mOrG * TILE_SIZE;
                rVA[currentVertex + 0].texCoords =
 Vector2f(0, 0 + verticalOffset);

                rVA[currentVertex + 1].texCoords =
 Vector2f(TILE_SIZE, 0 + verticalOffset);

                rVA[currentVertex + 2].texCoords =
 Vector2f(TILE_SIZE, TILE_SIZE + verticalOffset);

                rVA[currentVertex + 3].texCoords =
 Vector2f(0, TILE_SIZE + verticalOffset);
            }
// Position ready for the next for vertices
            currentVertex = currentVertex + VERTS_IN_QUAD;
        }
```

```
        }
        return TILE_SIZE;
    }
```

The preceding highlighted code starts by seeding the random number generator with a formula that will be different in each pass through the loop. Then, the `mOrG` variable is initialized with a number between 0 and `TILE_TYPES`. This is just what we need to pick one of the tile types randomly.

`mOrG` stands for "mud or grass." The name is arbitrary.

Now, we declare and initialize a variable called `verticalOffset` by multiplying `mOrG` by `TileSize`. We now have a vertical reference point within the sprite sheet to the starting height of the randomly chosen texture for the current quad.

Now, we use a simple formula involving `TILE_SIZE` and `verticalOffset` to assign the precise coordinates of each corner of the texture to the appropriate vertex.

We can now put our new function to work in the game engine.

Using the background

We have done the tricky stuff already, so this will be simple. There are three steps, as follows:

1. Create a `VertexArray`.
2. Initialize it after leveling up each wave.

3. Draw it in each frame.

Before we add the new code, the `ZombieArena.cpp` file needs to know about the new `ZombieArena.h` file. Add the following `include` directive to the top of the `ZombieArena.cpp` file:

```
#include "ZombieArena.h"
```

Now, add the following highlighted code to declare a `VertexArray` instance called `background` and load the `background_sheet.png` file as a texture:

```
// Create an instance of the Player class
Player player;
// The boundaries of the arena
IntRect arena;
// Create the background
VertexArray background;
// Load the texture for our background vertex array
Texture textureBackground;
textureBackground.loadFromFile("graphics/background_sheet.png")
// The main game loop
while (window.isOpen())
```

Add the following code to call the `createBackground` function, passing in `background` as a reference and `arena` by value. Notice that, in the highlighted code, we have also modified the way that we initialize the `tileSize` variable. Add the highlighted code exactly as shown:

```
if (state == State::PLAYING)
{
    // Prepare the level
// We will modify the next two lines later
    arena.width = 500;
    arena.height = 500;
```

```
        arena.left = 0;
        arena.top = 0;
    // Pass the vertex array by reference
    // to the createBackground function
    int tileSize = createBackground(background, arena);
     // We will modify this line of code later
    // int tileSize = 50;
     // Spawn the player in the middle of the arena
        player.spawn(arena, resolution, tileSize);
        // Reset the clock so there isn't a frame jump
        clock.restart();
    }
```

Note that we have replaced the `int tileSize = 50` lines of code because we get the value directly from the return value of the `createBackground` function.

For the sake of future code clarity, you should delete the `int tileSize = 50` lines of code and its related comment. I just commented it out to give the new code a clearer context.

Finally, it is time to do the drawing. This is really simple. All we do is call `window.draw` and pass the `VertexArray` instance, along with the `textureBackground` texture's memory address:

```
    /*
     **************
     Draw the scene
     **************
     */
    if (state == State::PLAYING)
    {
        window.clear();
        // Set the mainView to be displayed in the window
```

```
// And draw everything related to it
    window.setView(mainView);
 // Draw the background
    window.draw(background, &textureBackground);
// Draw the player
    window.draw(player.getSprite());
}
```

If you are wondering what is going on with the odd-looking `&` sign in front of `textureBackground`, then all will be made clear in the next chapter.

You can now run the game. You will see the following output. Remember to press *Enter* and select a number key to get past our temporarily invisible menu:

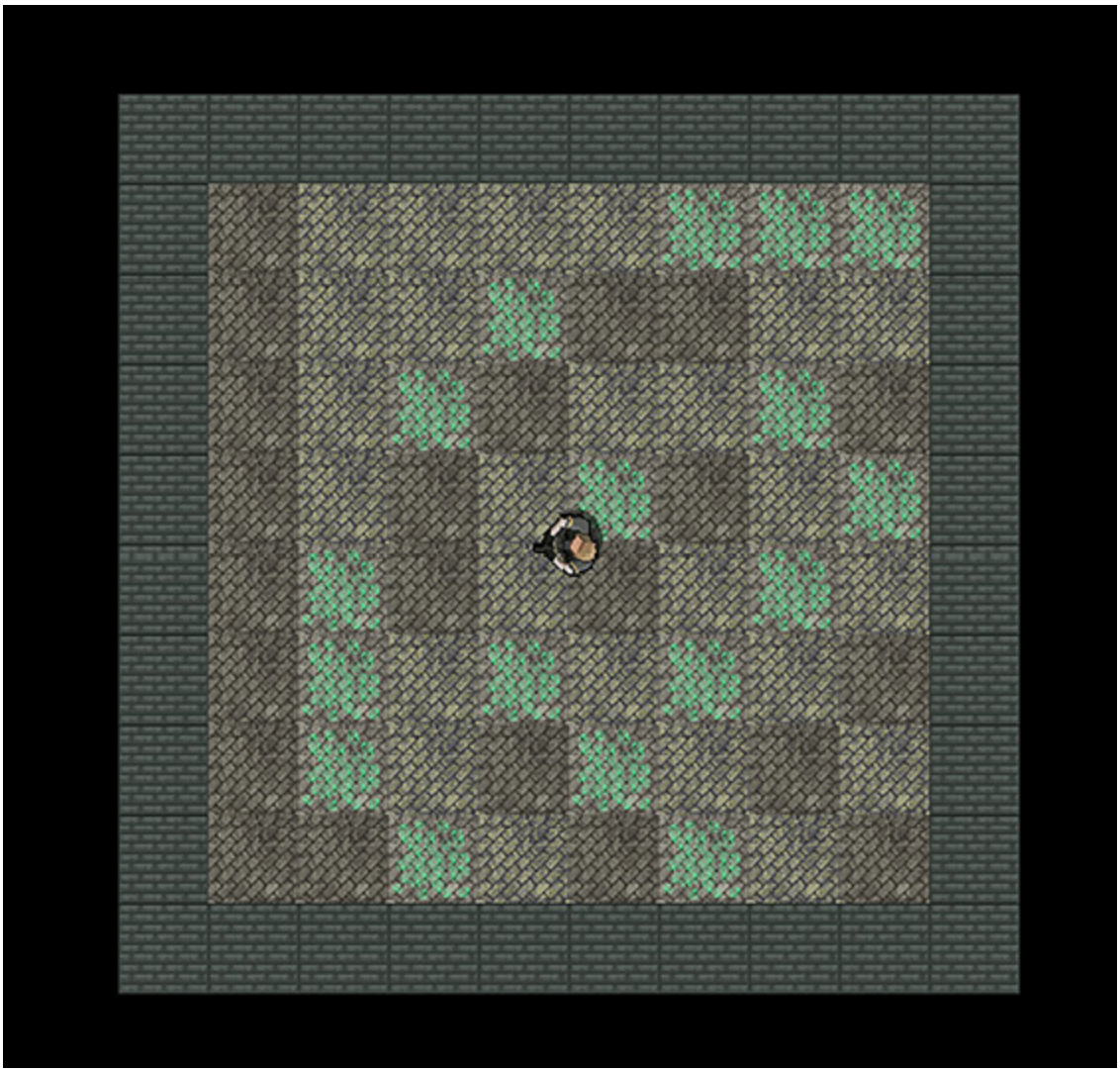


Figure 9.3: Using the background

Here, note how the player's sprite glides and rotates smoothly within the arena's confines. Although the current code in the `main` function draws a small arena, the `CreateBackground` function can create an arena of any size. We will see arenas bigger than the screen in *Chapter 14*, *Sound Effects, File I/O, and Finishing the Game*.

Summary

In this chapter, we discovered C++ references, which are special variables that act as aliases for other variables. When we pass a variable by reference instead of by value, then anything we do on the reference happens to the variable back in the calling function.

We also learned about vertex arrays and created a vertex array full of quads to draw the tiles from a sprite sheet as a background.

The elephant in the room, of course, is that our zombie game doesn't have any zombies. We'll fix that in the next chapter by learning about C++ pointers and the **Standard Template Library** (**STL**).

Frequently asked questions

Here are some questions that might be on your mind:

Q) Can you summarize these references again?

A) You must initialize a reference immediately, and it cannot be changed to reference another variable. Use references with functions so you are not working on a copy. This is good for efficiency because it avoids making copies and helps us abstract our code into functions more easily.

Q) Is there an easy way to remember the main benefit of using references?

A) To help you remember what a reference is used for, consider this short rhyme:

Moving large objects can make our games choppy,

passing by reference is faster than copy.

10

Pointers, the Standard Template Library, and Texture Management

We will be learning a lot as well as getting plenty done in terms of the game in this chapter. We will first learn about the fundamental C++ topic of **pointers**. Pointers are variables that hold a memory address. Typically, a pointer will hold the memory address of another variable. This sounds a bit like a reference, but we will see how they are much more powerful and use a pointer to handle an ever-expanding horde of zombies.

We will also learn about the **Standard Template Library** (**STL**), which is a collection of classes that allow us to quickly and easily implement common data management techniques.

In this chapter, we will cover the following topics:

- Learning about pointers
- Learning about the Standard Template Library

Learning about pointers

Pointers can be the cause of frustration while learning to code C++. However, the concept is simple.

A **pointer** is a variable that holds a memory address.

That's it! There's nothing to be concerned about. What probably causes frustration for beginners is the syntax—the code we use to handle pointers. We will step through each part of the code for using pointers. You can then begin the ongoing process of mastering them. Later, in the final project, we will learn about smart pointers, which in some ways simplify what we are about to learn but are less flexible.

In this section, we will actually learn more about pointers than we need to for this project. In the next project, we will make greater use of pointers. Despite this, we will only scratch the surface of this topic.

Rarely do I suggest that memorizing facts, figures, or syntax is the best way to learn. However, memorizing the brief but crucial syntax related to pointers might be worthwhile. This will ensure that the information sinks so deep into our brains that we can never forget it. We can then talk about why we would need pointers at all and examine their relationship to references. A pointer analogy might help:

If a variable type is a house and its contents are the value it holds, then a pointer is the address of the house.

In the previous chapter, while discussing references, we learned that when we pass values to, or return values from, a function, we are actually making

a completely new variable type, but it's exactly the same as the previous one. We are making a copy of the value that's passed to or from a function.

At this point, pointers are probably starting to sound a bit like references. That's because they are a bit like references. Pointers, however, are much more flexible and powerful and have their own special and unique uses. These special and unique uses require a special and unique syntax. Let's look at that first.

Pointer syntax

There are two main operators associated with pointers. The first is the **address of** operator:

```
&
```

The second is the **dereference** operator:

```
*
```

We will now look at the different ways in which we can use these operators with pointers.

The first thing you will notice is that the address of the operator is the same as the reference operator. To add to the woes of an aspiring C++ game programmer, the operators do different things in different contexts. Knowing this from the outset is valuable. If you are staring at some code involving pointers and it seems like you are going mad, know this:

You are perfectly sane! You just need to look at the detail of the context.

Now, you know that if something isn't clear and immediately obvious, it is not your fault. Pointers are not clear and immediately obvious, but looking carefully at the context will reveal what is going on.

Armed with the knowledge that you need to pay more attention to pointers than to previous syntax, as well as what the two operators are (**address of** and **dereference**), we can now start to look at some real pointer code.

Make sure you have memorized the two operators before proceeding.

Declaring a pointer

To declare a new pointer, we use the dereference operator, along with the type of variable the pointer will be holding the address of. Take a look at the following code before we talk about pointers some more:

```
// Declare a pointer to hold
// the address of a variable of type int
int* pHealth;
```

The preceding code declares a new pointer called `pHealth` that can hold the address of a variable of the `int` type. Notice I said *can* hold a variable of the `int` type. Like other variables, a pointer also needs to be initialized with a value to make proper use of it.

The name `pHealth`, just like other variables, is arbitrary.

It is common practice to prefix the names of variables that are pointers with a p. It is then much easier to remember

when we are dealing with a pointer and can then distinguish them from regular variables.

The white space that's used around the dereference operator is optional because C++ rarely cares about spaces in syntax. However, it's recommended because it aids readability. Look at the following three lines of code that all do the same thing.

We have just seen the following format in the previous example, with the dereference operator next to the type:

```
int* pHealth;
```

The following code shows white space on either side of the dereference operator:

```
int * pHealth;
```

The following code shows the dereference operator next to the name of the pointer:

```
int *pHealth;
```

It is worth being aware of these possibilities so that when you read code, perhaps on the web, you will understand they are all the same. In this book, we will always use the first option with the dereference operator next to the type.

Just like a regular variable can only successfully contain data of the appropriate type, a pointer should only hold the address of a variable of the

appropriate type.

A pointer to the `int` type should not hold the address of a `String`, `Zombie`, `Player`, `Sprite`, `float`, or any other type, except `int`.

Let's see how we can initialize our pointers.

Initializing a pointer

Next, we will see how we can get the address of a variable into a pointer. Take a look at the following code:

```
// A regular int variable called health
int health = 5;
// Declare a pointer to hold the address of
// a variable of type int
int* pHealth;
// Initialize pHealth to hold the address of health,
// using the "address of" operator
pHealth = &health;
```

In the previous code, we declare an `int` variable called `health` and initialize it to `5`. It makes sense, although we have never discussed it before, that this variable must be somewhere in our computer's memory. It must have a memory address.

We can access this address using the **address of** operator. Look closely at the last line of the previous code. We initialize `pHealth` with the address of `health`, like this:

```
    pHealth = &health;
```

Our `pHealth` pointer now holds the address of the regular `int`, `health`.

In C++ terminology, we say that `pHealth` points to health.

We can use `pHealth` by passing it to a function so that the function can work on `health`, just like we did with references.

There would be no reason for pointers if that was all we were going to do with them, so let's take a look at reinitializing them.

Reinitializing pointers

A pointer, unlike a reference, can be reinitialized to point to a different address. Look at the following code:

```
// A regular int variable called health
int health = 5;
int score = 0;
// Declare a pointer to hold the address
// of a variable of type int
int* pHealth;
// Initialize pHealth to hold the address of health
pHealth = &health;
// Re-initialize pHealth to hold the address of score
pHealth = &score;
```

Now, `pHealth` points to the `int` variable, `score`.

Of course, the name of our pointer, `pHealth`, is now ambiguous and should perhaps have been called `pIntPointer`. The key thing to understand here is that we *can* do this reassignment.

At this stage, we haven't actually used a pointer for anything other than simply pointing (holding a memory address). Let's see how we can access

the value stored at the address that's pointed to by a pointer. This will make them genuinely useful.

Dereferencing a pointer

We know that a pointer holds an address in memory. If we were to output this address in our game, perhaps in our HUD, after it has been declared and initialized, it might look something like this: `9876`.

It is just a value – a value that represents an address in memory. On different operating systems and hardware types, the range of these values will vary. In the context of this book, we never need to manipulate an address directly. We only care about what the value stored at the address that is pointed to is.

The actual addresses used by variables are determined when the game is executed (at runtime) and so there is no way of knowing the address of a variable and hence the value stored in a pointer while we are coding the game.

We can access the value stored at the address that's pointed to by a pointer by using the **dereference** operator:

```
*
```

Yes, this is the exact same symbol we use to declare our pointers. Context is important. The following code manipulates some variables directly and by using a pointer. Try and follow along and then we will go through it:

Warning! The code that follows is pointless (pun intended).

It just demonstrates using pointers.

```
// Some regular int variables
int score = 0;
int hiScore = 10;
// Declare 2 pointers to hold the addresses of int
int* pIntPointer1;
int* pIntPointer2;
// Initialize pIntPointer1 to hold the address of score
pIntPointer1 = &score;
// Initialize pIntPointer2 to hold the address of hiScore
pIntPointer2 = &hiScore;
// Add 10 to score directly
score += 10;
// Score now equals 10
// Add 10 to score using pIntPointer1
*pIntPointer1 += 10;
// score now equals 20. A new high score
// Assign the new hi score to hiScore using only pointers
*pIntPointer2 = *pIntPointer1;
// hiScore and score both equal 20
```

In the previous code, we declare two `int` variables, `score` and `hiScore`. We then initialize them with the values `0` and `10`, respectively. Next, we declare two pointers to `int`. These are `pIntPointer1` and `pIntPointer2`. We initialize them in the same step as declaring them to hold the addresses of (point to) the `score` and `hiScore` variables, respectively.

Following on, we add `10` to `score` in the usual way, `score += 10`. Then, we can see that by using the dereference operator on a pointer, we can access the value stored at the address they point to. The following code changed the value stored by the variable that's pointed to by `pIntPointer1`:

```
// Add 10 to score using pIntPointer1
*pIntPointer1 += 10;
```

```
// score now equals 20, A new high score
```

The last part of the preceding code dereferences both pointers to assign the value that's pointed to by `pIntPointer1` as the value that's pointed to by `pIntPointer2`:

```
// Assign the new hi-score to hiScore with only pointers
*pIntPointer2 = *pIntPointer1;
// hiScore and score both equal 20
```

Both `score` and `hiScore` are now equal to `20`.

Pointers are versatile and powerful

We can do so much more with pointers. Here are just a few useful things we can do.

Dynamically allocated memory

All the pointers we have seen so far point to memory addresses that have a scope limited only to the function they are created in. So, if we declare and initialize a pointer to a local variable, when the function returns, the pointer, the local variable, and the memory address will be gone. They are out of scope.

Up until now, we have been using a fixed amount of memory that is decided in advance of the game being executed. Furthermore, the memory we have been using is controlled by the operating system, and variables are lost and created as we call and return from functions. What we need is a way to use memory that is always in scope until we are finished with it. We want to have access to memory we can call our own and take responsibility for.

When we declare variables (including pointers), they are in an area of memory known as the **stack**. We discussed how the stack works by adding and removing functions and their related parameters and local variables in *Chapter 4*. There is another area of memory that, although allocated and controlled by the operating system, can be allocated at runtime. This other area of memory is called the **heap**.

Memory on the heap does not have scope to a specific function. Returning from a function does not delete the memory on the heap.

This gives us great power. With access to memory that is only limited by the resources of the computer our game is running on, we can plan games with huge amounts of objects. In our case, we want a vast horde of zombies. As Spiderman's uncle wouldn't hesitate to remind us, however, "with great power comes great responsibility."

Let's look at how we can use pointers to take advantage of the memory on the heap and how we can release that memory back to the operating system when we are finished with it.

To create a pointer that points to a value on the heap, we need a pointer:

```
int* pToInt = nullptr;
```

In the previous line of code, we declare a pointer in the same way we have seen before, but since we are not initializing it to point to a variable, we initialize it to `nullptr`. We do this because it is good practice. Consider **dereferencing** a pointer (changing a value at the address it points to) when

you don't even know what it is pointing to. It would be the programming equivalent of going to the shooting range, blindfolding someone, spinning them around, and telling them to shoot. By pointing a pointer to nothing (`nullptr`), we can't do any harm with it.

When we are ready to request memory on the heap, we use the `new` keyword, as shown in the following line of code:

```
pToInt = new int;
```

`pToInt` now holds the memory address of space on the heap that is just the right size to hold an `int` value.

Any allocated memory is returned when the program ends. It is, however, important to realize that this memory will never be freed (within the execution of our game) unless we free it. If we continue to take memory from the heap without giving it back, eventually it will run out and the game will crash.

It is unlikely that we would ever run out of memory by occasionally taking `int`-sized chunks of the heap. But if our program has a function or loop that requests memory and this function or loop is executed regularly throughout the game, eventually the game will slow and then crash. Furthermore, if we allocate lots of objects on the heap and don't manage them correctly, then this situation can happen quite quickly.

The following line of code hands back (deletes) the memory on the heap that was previously pointed to by `pToInt`:

```
delete pToInt;
```

Now, the memory that was previously pointed to by `pToInt` is no longer ours to do what we like with; we must take precautions. Although the memory has been handed back to the operating system, `pToInt` still holds the address of this memory, which no longer belongs to us.

The following line of code ensures that `pToInt` can't be used to attempt to manipulate or access this memory:

```
pToInt = nullptr;
```

If a pointer points to an address that is invalid, it is called a **wild** or **dangling** pointer. If you attempt to dereference a dangling pointer and you are lucky, the game will crash, and you will get a **memory access violation error**. If you are unlucky, you will create a bug that will be incredibly difficult to find. Furthermore, if we use memory on the heap that will persist beyond the life of a function, we must make sure to keep a pointer to it or we will have leaked memory. That is, the memory will remain allocated, but we will have lost access to it. C++ smart pointers avoid these possibilities and are often the most appropriate choice, but it is hard to learn about smart pointers without first understanding regular pointers. Furthermore, there are things you can only do with a regular pointer.

Now, we can declare pointers and point them to newly allocated memory on the heap. We can manipulate and access the memory they point to by dereferencing them. We can also return memory to the heap when we are done with it, and we also know how to avoid having a dangling pointer.

Let's look at some more advantages of pointers.

Passing a pointer to a function

In order to pass a pointer to a function, we need to write a function that has a pointer in the prototype, like in the following code:

```
void myFunction(int *pInt)
{
   // Dereference and increment the value stored
   // at the address pointed to by the pointer
   *pInt ++
   return;
}
```

The preceding function simply dereferences the pointer and adds `1` to the value stored at the pointed-to address.

Now, we can use that function and pass the address of a variable or another pointer to a variable explicitly:

```
int someInt = 10;
int* pToInt = &someInt;
myFunction(&someInt);
// someInt now equals 11
myFunction(pToInt);
// someInt now equals 12
```

As shown in the previous code, within the function, we are manipulating the variable from the calling code and can do so using the address of a variable or a pointer to that variable, since both actions amount to the same thing.

Pointers can also point to instances of a class.

Declaring and using a pointer to an object

Pointers are not just for regular variables. We can also declare pointers to user-defined types such as our classes. This is how we would declare a pointer to an object of the `Player` type:

```
Player player;
Player* pPlayer = &Player;
```

We can even access the member functions of a `Player` object directly from the pointer, as shown in the following code:

```
// Call a member function of the player class
pPlayer->moveLeft()
```

Notice the subtle but vital difference: accessing a function with a pointer to an object rather than an object directly uses the -> operator.

The `->` operator in C++ is called the **member access operator** or sometimes simply the **arrow operator**. It is used to access members of a class through a pointer to that class. The `->` operator is a shorthand notation for dereferencing a pointer to an object and accessing a member of the object simultaneously.

We won't need to use pointers to objects in this project, but we will explore them more carefully before we do, which will be in the final project. Let's go over one more new pointer topic before we talk about something completely new.

Pointers and arrays

Arrays and pointers have something in common. An array's name is a memory address. More specifically, the name of an array is the memory address of the first element in that array. To put this yet another way, an array name points to the first element of an array. The best way to understand this is to read on and look at the following example.

We can create a pointer to the type that an array holds and then use the pointer in the same way using exactly the same syntax that we would use for the array:

```
// Declare an array of ints
int arrayOfInts[100];
// Declare a pointer to int and initialize it
// with the address of the first
// element of the array, arrayOfInts
int* pToIntArray = arrayOfInts;
// Use pToIntArray just as you would arrayOfInts
arrayOfInts[0] = 999;
// First element of arrayOfInts now equals 999
pToIntArray[0] = 0;
// First element of arrayOfInts now equals 0
```

This also means that a function that has a prototype that accepts a pointer also accepts arrays of the type the pointer is pointing to. We will use this fact when we build our ever-increasing horde of zombies.

Regarding the relationship between pointers and references, the compiler actually uses pointers when implementing our references. This means that references are just a handy tool (that uses pointers "under the hood"). You could think of a reference as an automatic gearbox that is fine and convenient for driving around town, whereas pointers are a manual gearbox – more complicated, but with the correct use, they can provide better results/performance/flexibility.

Summary of pointers

Pointers are a bit fiddly at times. In fact, our discussion of pointers was only an introduction to the subject. The only way to get comfortable with them is to use them as much as possible. All you need to understand about pointers in order to complete this project is the following:

- Pointers are variables that store a memory address.
- We can pass pointers to functions to directly manipulate values from the calling function's scope, within the called function.
- Array names hold the memory address of the first element. We can pass this address as a pointer because that is exactly what it is.
- We can use pointers to point to memory on the heap. This means we can dynamically allocate large amounts of memory while the game is running.

There are yet more ways to use pointers. We will learn about **smart pointers** in the final project, once we have got used to using regular pointers.

There is just one more topic to cover before we can start coding the Zombie Arena project again.

Learning about the Standard Template Library

The **Standard Template Library** (**STL**) is a collection of data containers and ways to manipulate the data we put in those containers. If we want to be more specific, it is a way to store and manipulate different types of C++ variables and classes.

We can think of the different containers as customized and more advanced arrays. The STL is part of C++. It is not an optional thing that needs to be set up like SFML.

The STL is part of C++ because its containers and the code that manipulates them are fundamental to many types of code that many apps will need to use.

In short, the STL implements code that we and just about every C++ programmer is almost bound to need, at least at some point, and probably quite regularly.

If we were to write our own code to contain and manage our data, then it is unlikely we would write it as efficiently as the people who wrote the STL.

So, by using the STL, we guarantee that we are using the best-written code possible to manage our data. Even SFML uses the STL. For example, under the hood, the `VertexArray` class uses the STL.

All we need to do is choose the right type of container from those that are available. The types of containers that are available through the STL

include the following:

- **Vector**: This is like an array with boosters. It handles dynamic resizing, sorting, and searching. This is probably the most useful container. We will look at some vector code next.
- **List**: A container that allows for the ordering of the data.
- **Map**: An associative container that allows the user to store data as key/value pairs. This is where one piece of data is the "key" to finding the other piece. A map can also grow and shrink, as well as being searched. We will learn about maps after vectors and then go on to use a map.
- **Set**: A container that guarantees that every element is unique.

In the Zombie Arena game, we will use a map.

If you want a glimpse into the kind of complexity that the STL spares us, then take a look at this tutorial, which implements the kind of thing that a list would do. Note that the tutorial implements only the very simplest bare-bones implementation of a list: `http://www.sanfoundry.com/cpp-program-implement-single-linked-list/`.

We can easily see that we will save a lot of time and end up with a better game if we explore the STL. Let's take a closer look at how to use a `vector` instance, and then we will look at map as well as see how map will be useful to us in the Zombie Arena game.

What is a vector?

A **vector** in C++ is a dynamic array that allows us to store and manipulate a collection of elements. It provides a flexible and resizable container, similar to an array but with additional features that make it a powerful tool for managing collections of data.

Declaring a vector

To declare a vector, we use the `vector` template class from the **Standard Template Library** (**STL**). Here's an example of declaring a vector of integers:

```
// Add the vector header to the project
#include <vector>
vector<int> numbers;
```

In this example, `numbers` is a vector that can store integers. However, like arrays, vectors can be used to store elements of any data type.

Adding data to a vector

Let's add some integers to our vector:

```
numbers.push_back(42);
numbers.push_back(73);
numbers.push_back(10);
```

Now, our vector numbers contain three integers: `42`, `73`, and `10`.

Accessing data in a vector

We can access elements in a vector using the same array-like syntax:

```
int firstNumber = numbers[0]; // Access the first element (42)
int secondNumber = numbers[1]; // Access the second element (73
```

And we can remove data from a vector.

Removing data from a vector

Removing elements from a vector can be done using various methods. For example, to remove the first element:

```
numbers.erase(numbers.begin());
```

Now, `numbers` contains only two elements: `73` and `10`. This works because `numbers.begin` points to the first element and the `erase` function does exactly what the name suggests. All these functions are available because `numbers` is an instance of `vector`.

Checking the size of a vector

To find out how many elements are in a vector, we can use the `size` method:

```
int size = numbers.size(); // Size is now 2
```

The preceding code uses the `size` function, which returns the number of elements in a vector and stores the result in the `int` variable `size`.

Looping/iterating through the elements of a vector

We can use a loop to iterate through all the elements of a vector. Here's an example using a regular `for` loop:

```
for (vector<int>::iterator it = numbers.begin(); it != numbers.
{
    *it += 1; // Increment each element by 1
}
```

However, we can simplify this using the `auto` keyword:

```
for (auto it = numbers.begin(); it != numbers.end(); ++it)
{
    *it += 1; // Increment each element by 1
}
```

The `auto` keyword helps reduce verbosity by allowing the compiler to deduce the type for us. The type `vector<int>::iterator it` is the loop variable that is initialized to `numbers.begin()`. All the time this variable is not equal to `numbers.end`, we keep incrementing with `it++`. As we will see when we talk about maps, the format of these loops is quite flexible. This concise syntax improves code maintainability, as programmers no longer need to explicitly specify complex iterator types, resulting in cleaner and more intuitive loop structures, which is definitely a benefit to them.

Vectors are versatile and widely used in C++ for their dynamic resizing capabilities and straightforward syntax. They provide a convenient way to manage collections of data efficiently. In the final project, we will put vectors to work with a vector of game objects. But first, let's look at maps.

What is a map?

A `map` is a container that is dynamically resizable. We can add and remove elements with ease. What makes the `map` class special compared to the other containers in the STL is the way that we access the data within it.

The data in a `map` instance is stored in pairs. Consider a situation where you log in to an account, perhaps with a username and password. A `map` would be perfect for looking up the username and then checking the value of the associated password.

A `map` would also be just right for things such as account names and numbers, or perhaps company names and share prices.

Note that when we use `map` from the STL, we decide the type of values that form the key-value pairs. The values could be `string` instances and `int` instances, such as account numbers; `string` instances and other `string` instances, such as usernames and passwords; or user-defined types such as objects.

What follows is some real code to make us familiar with `map`.

Declaring a map

This is how we could declare a `map`:

```
map<string, int> accounts;
```

The previous line of code declares a new `map` called `accounts` that has a key of `string` objects, each of which will refer to a value that is an `int`.

We can now store key-value pairs of the `string` type that refer to values of the `int` type. We will see how we can do this next.

Adding data to a map

Let's go ahead and add a key-value pair to `accounts`:

```
accounts["John"] = 1234567;
```

Now, there is an entry in the `map` that can be accessed using the key of `John`. The following code adds two more entries to the `accounts` `map`:

```
accounts["Smit"] = 7654321;
accounts["Larissa"] = 8866772;
```

Our `map` has three entries in it. Let's see how we can access the account numbers.

Finding data in a map

We would access the data in the same way that we added it: by using the key. As an example, we could assign the value stored by the `Smit` key to a new `int`, `accountNumber`, like this:

```
int accountNumber = accounts["Smit"];
```

The `int` variable, `accountNumber`, now stores the value `7654321`. We can do anything to a value stored in a `map` instance that we can do to that type.

Removing data from a map

Taking values out of our `map` is also straightforward. The following line of code removes the key, `John`, and its associated value:

```
accounts.erase("John");
```

Let's look at a few more things we can do with a `map`.

Checking the size of a map

We might like to know how many key-value pairs we have in our `map`. The following line of code does just that:

```
int size = accounts.size();
```

The `int` variable, `size`, now holds the value of `2`. This is because `accounts` holds values for `Smit` and `Larissa`, because we deleted `John`.

Checking for keys in a map

The most relevant feature of `map` is its ability to find a value using a key. We can test for the presence or otherwise of a specific key like this:

```
if(accounts.find("John") != accounts.end())
{
    // This code won't run because John was erased
}
if(accounts.find("Smit") != accounts.end())
{
    // This code will run because Smit is in the map
}
```

In the previous code, the `!= accounts.end` value is used to determine when a key does or doesn't exist. If the searched-for key is not present in the `map`, then `accounts.end` will be the result of the `if` statement.

Let's see how we can test or use all the values in a `map` by looping through a map.

Looping/iterating through the key-value pairs of a map

We have seen how we can use a `for` loop to loop/iterate through all the values of an array. But what if we want to do something like this to a map?

The following code shows how we could loop through each key-value pair of the account's `map` and add one to each of the account numbers:

```
for (map<string,int>::iterator it = accounts.begin();
    it != accounts.end();
    ++ it)
{
    it->second += 1;
}
```

The condition of the `for` loop is probably the most interesting part of the previous code. The first part of the condition is the longest part. `map<string,int>::iterator it = accounts.begin()` is more understandable if we break it down.

`map<string,int>::iterator` is a type. We are declaring an `iterator` that's suitable for a `map` with key-value pairs of `string` and `int`.

The iterator's name is `it`. We assign the value that's returned by `accounts.begin()` to `it`. The iterator, `it`, now holds the first key-value pair from the `accounts` map.

The rest of the condition of the `for` loop works as follows. `it != accounts.end()` means the loop will continue until the end of the map is

reached, and `++it` simply steps to the next key-value pair in the map, each pass through the loop.

Inside the `for` loop, `it->second` accesses the value of the key-value pair and `+= 1` adds 1 to the value. Note that we can access the key (which is the first part of the key-value pair) with `it->first`.

You might have noticed that the syntax for setting up a loop through a map is quite verbose. C++ has a way to cut down on this verbosity.

The auto keyword

The code in the condition of the `for` loop was quite verbose – especially in terms of `map<string,int>::iterator`. As with the tutorials on `vector`, we use `auto` to reduce verbosity. Using the `auto` keyword, we can improve the previous code:

```
for (auto it = accounts.begin(); it != accounts.end(); ++ it)
{
    it->second += 1;
}
```

The `auto` keyword instructs the compiler to automatically deduce the type for us. This will be especially useful with the next class that we write.

STL summary

As with almost every C++ concept that we have covered in this book, the STL is a massive topic. Whole books have been written covering just the STL. At this point, however, we know enough to build a class that uses the STL `map` to store SFML `Texture` objects. We can then have textures that

can be retrieved/loaded by using the filename as the key of the key-value pair.

The reason why we would go to this extra level of complexity and not just carry on using the `Texture` class the same way as we have been so far will become apparent as we proceed.

Summary

In this chapter, we have covered pointers and discussed that they are variables that hold a memory address to a specific type of object. The full significance of this will begin to reveal itself as this book progresses and the power of pointers is revealed.

We also used pointers in order to create a huge horde of zombies that can be accessed using a pointer, which it turns out is also the same thing as the first element of an array.

We learned about the STL, and in particular the `map` class. We implemented a class that will store all our textures, as well as providing access to them.

In the next chapter, we will get to use what we have learned in this chapter. We will implement a horde of zombies using pointers and arrays and we will explore a neat way to handle textures for our sprites by using a map. We will also dig a little deeper into OOP and use a static function, which is a function of a class that can be called without an instance of the class.

Frequently asked questions

Here are some questions that might be on your mind:

Q) What's the difference between pointers and references?

A) Pointers are like references with boosters. Pointers can be changed to point to different variables (memory addresses), as well as point to dynamically allocated memory on the heap.

Q) What's the deal with arrays and pointers?

A) Arrays are really constant pointers to their first element.

11

Coding the TextureHolder Class and Building a Horde of Zombies

Now we understand the basics of the **Standard Template Library** (**STL**), we will be able to use that new knowledge to manage all the textures from the game because, if we have 1,000 zombies, we don't really want to load a copy of a zombie graphic into the GPU for each and every one.

We will also dig a little deeper into OOP and use a **static** function, which is a function of a class that can be called without an instance of the class. At the same time, we will see how we can design a class to ensure that only one instance can ever exist. This is ideal when we need to guarantee that different parts of our code will use the same data.

In this chapter, we will cover the following topics:

- Implementing the `TextureHolder` class
- Building a horde of zombies
- Using the `TextureHolder` class for all textures

Implementing the TextureHolder class

Thousands of zombies represent a new challenge. Loading, storing, and manipulating thousands of copies of three different zombie textures take up not only a lot of memory but also a lot of processing power. We will create a new type of class that overcomes this problem and allows us to store just one of each texture.

We will also code the class in such a way that there can only ever be one instance of it. This type of class is called a **singleton**.

Singleton is a design pattern. A design pattern is a way to structure our code that is proven to work.

Furthermore, we will also code the class so that it can be used anywhere in our game code directly through the class name, without access to an instance. This is a special type of class called a **static class**.

Coding the TextureHolder header file

Let's make a new header file. Right-click **Header Files** in the **Solution Explorer** and select **Add** | **New Item...**. In the **Add New Item** window, highlight (by left-clicking) **Header File (.h)**, and then in the **Name** field, type `TextureHolder.h`.

Add the code that follows into the `TextureHolder.h` file, and then we can discuss it:

```
#pragma once
#ifndef TEXTURE_HOLDER_H
#define TEXTURE_HOLDER_H
#include <SFML/Graphics.hpp>
#include <map>
using namespace sf;
```

```
using namespace std;
class TextureHolder
{
private:
    // A map container from the STL,
// that holds related pairs of String and Texture
    map<string, Texture> m_Textures;
    // A pointer of the same type as the class itself
// the one and only instance
static TextureHolder* m_s_Instance;
public:
    TextureHolder();
    static Texture& GetTexture(string const& filename);
};
#endif
```

In the previous code, notice that we have an `include` directive for `map` from the STL. We declare a `map` instance that holds the `string` type and the SFML `Texture` type, as well as the key-value pairs. The `map` is called `m_Textures`.

In the preceding code, this line follows on:

```
static TextureHolder* m_s_Instance;
```

The previous line of code is quite interesting. We are declaring a static pointer to an object of the `TextureHolder` type called `m_s_Instance`. This means that the `TextureHolder` class has an object that is the same type as itself. Not only that but, because it is static, it can be used through the class itself, without an instance of the class. When we code the related `.cpp` file, we will see how we can use this.

In the `public` part of the class, we have the prototype for the constructor function, `TextureHolder`. The constructor takes no arguments and, as usual, has no return type. This is the same as the default constructor. We are going

to override the default constructor with a definition that makes our singleton work how we want it to.

We have another function called `GetTexture`. Let's look at the signature again and analyze exactly what is happening:

```
static Texture& GetTexture(string const& filename);
```

First, notice that the function returns a reference to a `Texture`. This means that `GetTexture` will return a reference, which is efficient because it avoids making a copy of what could be a large graphic. Also, notice that the function is declared as `static`. This means that the function can be used without an instance of the class. The function takes a `string` as a constant reference as a parameter. The effect of this is two-fold. Firstly, the operation is efficient, and secondly, as the reference is constant, it can't be changed.

Next, we'll move on to coding the `TextureHolder` function definitions.

Coding the TextureHolder function definitions

Now, we can create a new `.cpp` file that will contain the function definition. This will allow us to see the reasons behind our new types of functions and variables. Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item...**. In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)**, and then in the **Name** field, type `TextureHolder.cpp`. Finally, click the **Add** button. We are now ready to code the class.

Add the following code, and then we can discuss it:

```
#include "TextureHolder.h"
// Include the "assert feature"
#include <assert.h>
TextureHolder* TextureHolder::m_s_Instance = nullptr;
TextureHolder::TextureHolder()
{
    assert(m_s_Instance == nullptr);
    m_s_Instance = this;
}
```

In the previous code, we initialize our pointer of the `TextureHolder` type to `nullptr`. In the constructor, `assert(m_s_Instance == nullptr)` ensures that `m_s_Instance` equals `nullptr`. If it doesn't, the game will exit execution. Then, `m_s_Instance = this` assigns the pointer to the `this` instance. Now, consider where this code is taking place. The code is in the constructor. The constructor is the way that we create instances of objects from classes. So, effectively, we now have a pointer to a `TextureHolder` that points to the one and only instance of itself.

Add the final part of the code to the `TextureHolder.cpp` file. There are more comments than code here. Examine the following code and read the comments as you add the code, and then we can go through it:

```
Texture& TextureHolder::GetTexture(string const& filename)
{
    // Get a reference to m_Textures using m_s_Instance
auto& m = m_s_Instance->m_Textures;
    // auto is the equivalent of map<string, Texture>
// Create an iterator to hold a key-value-pair (kvp)
// and search for the required kvp
// using the passed in file name
auto keyValuePair = m.find(filename);
    // auto is equivalent of map<string, Texture>::iterator
// Did we find a match?
if (keyValuePair != m.end())
    {
```

```
        // Yes
// Return the texture,
// the second part of the kvp, the texture
return keyValuePair->second;
    }
    else
    {
        // File name not found
// Create a new key value pair using the filename
auto& texture = m[filename];
        // Load the texture from file in the usual way
        texture.loadFromFile(filename);
        // Return the texture to the calling code
return texture;
    }
}
```

The first thing you will probably notice about the previous code is the `auto` keyword. The `auto` keyword was explained in the previous section.

If you want to know what the actual types that have been replaced by `auto` are, then look at the comments immediately after each use of `auto` in the previous code. You can also hover over the `auto` keyword in Visual Studio and see a tooltip showing the full type.

At the start of the code, we get a reference to `m_textures`. Then, we attempt to get an iterator to the key-value pair represented by the passed-in filename (`filename`). If we find a matching key, we return the texture with `return keyValuePair->second`. Otherwise, we add the texture to the `map` and then return it to the calling code.

Admittedly, the `TextureHolder` class introduced lots of new concepts (singletons, `static` functions, constant references, `this` and the `auto`

keyword,) and syntax. Add to this the fact that we have only just learned about pointers and the STL, and this section's code might have been a little daunting.

So, was it all worth it?

What have we achieved with TextureHolder?

The point is that now that we have this class, we can go wild using textures from wherever we like in our code and not worry about running out of memory or having access to any texture in a particular function or class. We will see how to use `TextureHolder` soon.

Building a horde of zombies

Now, we are armed with the `TextureHolder` class to make sure that our zombie textures are easily available as well as only loaded into the GPU once. Then, we can investigate creating a whole horde of them.

We will store zombies in an array. Since the process of building and spawning a horde of zombies involves quite a few lines of code, it is a good candidate for abstracting to a separate function. Soon, we will code the `CreateHorde` function but first, of course, we need a `Zombie` class.

Coding the Zombie.h file

The first step to building a class to represent a zombie is to code the member variables and function prototypes in a header file.

Right-click **Header Files** in the **Solution Explorer** and select **Add** | **New Item...**. In the **Add New Item** window, highlight (by left-clicking) **Header File (.h)**, and then in the **Name** field, type `Zombie.h`.

Add the following code to the `Zombie.h` file:

```
#pragma once
#include <SFML/Graphics.hpp>
using namespace sf;
class Zombie
{
private:
    // How fast is each zombie type?
const float BLOATER_SPEED = 40;
    const float CHASER_SPEED = 80;
    const float CRAWLER_SPEED = 20;
    // How tough is each zombie type
const float BLOATER_HEALTH = 5;
    const float CHASER_HEALTH = 1;
    const float CRAWLER_HEALTH = 3;
    // Make each zombie vary its speed slightly
const int MAX_VARRIANCE = 30;
    const int OFFSET = 101 - MAX_VARRIANCE;
    // Where is this zombie?
    Vector2f m_Position;
    // A sprite for the zombie
    Sprite m_Sprite;
    // How fast can this one run/crawl?
float m_Speed;
    // How much health has it got?
float m_Health;
    // Is it still alive?
bool m_Alive;

    // Public prototypes go here
};
```

The previous code declares all the private member variables of the `Zombie` class. At the top of the previous code, we have three constant variables to

hold the speed of each type of zombie: a very slow **Crawler**, a slightly faster **Bloater**, and a somewhat speedy **Chaser**. We can experiment with the value of these three constants to help balance the difficulty level of the game. It's also worth mentioning here that these three values are only used as a starting value for the speed of each zombie type. As we will see later in this chapter, we will vary the speed of every zombie by a small percentage from these values. This stops zombies of the same type from bunching up together as they pursue the player.

The next three constants determine the health level for each zombie type. Note that Bloaters are the toughest, followed by Crawlers. As a matter of balance, the Chaser zombies will be the easiest to kill.

Next, we have two more constants, `MAX_VARRIANCE` and `OFFSET`. These will help us determine the individual speed of each zombie. We will see exactly how when we code the `Zombie.cpp` file.

After these constants, we declare a bunch of variables that should look familiar because we had very similar variables in our `Player` class. The `m_Position`, `m_Sprite`, `m_Speed`, and `m_Health` variables are for what their names imply: the position, sprite, speed, and health of the zombie object, respectively.

Finally, in the preceding code, we declare a Boolean called `m_Alive`, which will be true when the zombie is alive and hunting, but false when its health gets to 0 and it is just a splurge of blood on our otherwise pretty background.

Now, we can complete the `Zombie.h` file. Add the function prototypes highlighted in the following code, and then we will talk about them:

```
 // Is it still alive?
bool m_Alive;
```

```
    // Public prototypes go here
public:

 // Handle when a bullet hits a zombie
 bool hit();
 // Find out if the zombie is alive
 bool isAlive();
 // Spawn a new zombie
 void spawn(float startX, float startY, int type, int seed);
 // Return a rectangle that is the position in the world
 FloatRect getPosition();
 // Get a copy of the sprite to draw
 Sprite getSprite();
 // Update the zombie each frame
 void update(float elapsedTime, Vector2f playerLocation);
};
```

In the previous code, there is a `hit` function, which we can call every time the zombie is hit by a bullet. The function can then take the necessary steps, such as taking health from the zombie (reducing the value of `m_Health`) or killing it (setting `m_Alive` to `false`).

The `isAlive` function returns a Boolean that lets the calling code know whether the zombie is alive or dead. We don't want to perform collision detection or remove health from the player for walking over a blood splat.

The `spawn` function takes a starting position, a type (Crawler, Bloater, or Chaser, represented by an `int`), as well as a seed to use in some random number generation that we will see in the next section.

Just like we have in the `Player` class, the `Zombie` class has `getPosition` and `getSprite` functions to get a rectangle that represents the space occupied by the zombie and the sprite that can be drawn in each frame.

The last prototype in the previous code is the `update` function. We could have probably guessed that it would receive the elapsed time since the last

frame, but also, notice that it receives a `Vector2f` vector called `playerLocation`. This vector will indeed be the exact coordinates of the center of the player. We will soon see how we can use this vector to chase after the player.

Now, we can code the function definitions in the `.cpp` file.

Coding the Zombie.cpp file

Next, we will code the functionality of the `Zombie` class – the function definitions.

Create a new `.cpp` file that will contain the function definitions. Right-click **Source Files** in the **Solution Explorer** and select **Add** | **New Item...**. In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)**, and then in the **Name** field, type `Zombie.cpp`. Finally, click the **Add** button. We are now ready to code the class.

Add the following code to the `Zombie.cpp` file:

```
#include "zombie.h"
#include "TextureHolder.h"
#include <cstdlib>
#include <ctime>
using namespace std;
```

First, we add the necessary `include` directives and then `using namespace std`. You might remember a few instances when we prefixed our object declarations with `std::`. This `using` directive means we don't need to do that for the code in this file.

Now, add the following code, which is the definition of the `spawn` function. Study the code once you have added it, and then we will discuss it:

```
void Zombie::spawn(float startX, float startY, int type, int see
{

    switch (type)
    {
    case 0:
        // Bloater
        m_Sprite = Sprite(TextureHolder::GetTexture(
            "graphics/bloater.png"));
        m_Speed = BLOATER_SPEED;
        m_Health = BLOATER_HEALTH;
        break;
    case 1:
        // Chaser
        m_Sprite = Sprite(TextureHolder::GetTexture(
            "graphics/chaser.png"));
        m_Speed = CHASER_SPEED;
        m_Health = CHASER_HEALTH;
        break;
    case 2:
        // Crawler
        m_Sprite = Sprite(TextureHolder::GetTexture(
            "graphics/crawler.png"));
        m_Speed = CRAWLER_SPEED;
        m_Health = CRAWLER_HEALTH;
        break;
    }
    // Modify the speed to make the zombie unique
// Every zombie is unique. Create a speed modifier
srand((int)time(0) * seed);
    // Somewhere between 80 and 100
float modifier = (rand() % MAX_VARRIANCE) + OFFSET;
    // Express this as a fraction of 1
    modifier /= 100; // Now equals between .7 and 1
    m_Speed *= modifier;

    // Initialize its location
    m_Position.x = startX;
    m_Position.y = startY;
    // Set its origin to its center
    m_Sprite.setOrigin(25, 25);
    // Set its position
```

```
    m_Sprite.setPosition(m_Position);
}
```

The first thing the function does is switch paths of execution based on the `int` value, which is passed in as a parameter. Within the `switch` block, there is a `case` for each type of zombie. Depending on the type of zombie, the appropriate texture, speed, and health are initialized to the relevant member variables.

We could have used an enumeration for the different types of zombie. Feel free to upgrade your code when the project is finished.

Of interest here is that we use the static `TextureHolder::GetTexture` function to assign the texture. This means that no matter how many zombies we spawn, there will be a maximum of three textures in the memory of the GPU.

The next three lines of code (excluding comments) do the following:

- Seed the random number generator with the `seed` variable that was passed in as a parameter.
- Declare and initialize the `modifier` variable using the `rand` function and the `MAX_VARRIANCE` and `OFFSET` constants. The result is a fraction between 0 and 1, which can be used to make each zombie's speed unique. The reason we want to do this is so that the zombies don't bunch up on top of each other too much.
- We can now multiply `m_Speed` by `modifier` and we will have a zombie whose speed is within the `MAX_VARRIANCE` percentage of the constant

defined for this type of zombie's speed.

After we have resolved the speed, we assign the passed-in position held in `startX` and `startY` to `m_Position.x` and `m_Position.y`, respectively.

The last two lines of code in the previous listing set the origin of the sprite to the center and use the `m_Position` vector to set the position of the sprite.

Now, add the following code for the `hit` function to the `Zombie.cpp` file:

```
bool Zombie::hit()
{
    m_Health--;
    if (m_Health < 0)
    {
        // dead
        m_Alive = false;
        m_Sprite.setTexture(TextureHolder::GetTexture(
            "graphics/blood.png"));
        return true;
    }
    // injured but not dead yet
return false;
}
```

The `hit` function is nice and simple: reduce `m_Health` by 1 and then check whether `m_Health` is below 0.

If it is below 0, then it sets `m_Alive` to `false`, swaps the zombie's texture for a blood splat, and returns `true` to the calling code so that it knows the zombie is now dead. If the zombie has survived, the hit returns `false`.

Add the following three getter functions, which just return a value to the calling code:

```
bool Zombie::isAlive()
{
```

```
    return m_Alive;
}
FloatRect Zombie::getPosition()
{
    return m_Sprite.getGlobalBounds();
}
Sprite Zombie::getSprite()
{
    return m_Sprite;
}
```

The previous three functions are quite self-explanatory, perhaps with the exception of the `getPosition` function, which uses the `m_Sprite.getLocalBounds` function to get the `FloatRect` instance, which is then returned to the calling code.

Finally, for the `Zombie` class, we need to add the code for the `update` function. Look closely at the following code as you add it, and then we will go through it:

```
void Zombie::update(float elapsedTime,
    Vector2f playerLocation)
{
    float playerX = playerLocation.x;
    float playerY = playerLocation.y;
    // Update the zombie position variables
if (playerX > m_Position.x)
    {
        m_Position.x = m_Position.x +
            m_Speed * elapsedTime;
    }
    if (playerY > m_Position.y)
    {
        m_Position.y = m_Position.y +
            m_Speed * elapsedTime;
    }

    if (playerX < m_Position.x)
```

```
        {
            m_Position.x = m_Position.x -
                m_Speed * elapsedTime;
        }
        if (playerY < m_Position.y)
        {
            m_Position.y = m_Position.y -
                m_Speed * elapsedTime;
        }
        // Move the sprite
        m_Sprite.setPosition(m_Position);
        // Face the sprite in the correct direction
    float angle = (atan2(playerY - m_Position.y,
            playerX - m_Position.x)
            * 180) / 3.141;
        m_Sprite.setRotation(angle);
    }
```

In the preceding code, we copy `playerLocation.x` and `playerLocation.y` into the local variables called `playerX` and `playerY`.

Next, there are four `if` statements. They test to see whether the zombie is to the left, right, above, or below the current player's position.

These four `if` statements, when they evaluate to `true`, adjust the zombie's `m_Position.x` and `m_Position.y` values appropriately using the usual formula – that is, speed multiplied by time since the last frame. More specifically, the code is `m_Speed * elapsedTime`.

After the four `if` statements, `m_Sprite` is moved to its new location.

We then use the same calculation we previously used with the player and the mouse pointer but, this time, we do so for the zombie and the player. This calculation finds the angle that's needed to face the zombie toward the player.

Finally, for this function and the class, we call `m_Sprite.setRotation` to actually rotate the zombie sprite. Remember that this function will be called for every zombie (that is alive) in every frame of the game.

But we want a whole horde of zombies.

Using the Zombie class to create a horde

Now that we have a class to create a living, attacking, and killable zombie, we want to spawn a whole horde of them.

To achieve this, we will write a separate function and we will use a pointer so that we can refer to our horde that will be declared in `main` but configured in a different scope.

Open the `ZombieArena.h` file in Visual Studio and add the following highlighted lines of code:

```
#pragma once
#include "Zombie.h"
using namespace sf;
int createBackground(VertexArray& rVA, IntRect arena);
Zombie* createHorde(int numZombies, IntRect arena);
```

Now that we have a prototype, we can code the function definition.

Create a new `.cpp` file that will contain the function definition. Right-click **Source Files** in the **Solution Explorer** and select **Add** | **New Item...**. In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)**, and then in the **Name** field, type `CreateHorde.cpp`. Finally, click the **Add** button.

Add the following code to the `CreateHorde.cpp` file and study it. Afterward, we will break it down into chunks and discuss it:

```
#include "ZombieArena.h"
#include "Zombie.h"
Zombie* createHorde(int numZombies, IntRect arena)
{
    Zombie* zombies = new Zombie[numZombies];
    int maxY = arena.height - 20;
    int minY = arena.top + 20;
    int maxX = arena.width - 20;
    int minX = arena.left + 20;
    for (int i = 0; i < numZombies; i++)
    {

        // Which side should the zombie spawn
srand((int)time(0) * i);
        int side = (rand() % 4);
        float x, y;
        switch (side)
        {
        case 0:
            // left
            x = minX;
            y = (rand() % maxY) + minY;
            break;
        case 1:
            // right
            x = maxX;
            y = (rand() % maxY) + minY;
            break;
        case 2:
            // top
            x = (rand() % maxX) + minX;
            y = minY;
            break;
        case 3:
            // bottom
            x = (rand() % maxX) + minX;
            y = maxY;
            break;
```

```
            }
            // Bloater, crawler or runner
    srand((int)time(0) * i * 2);
            int type = (rand() % 3);
            // Spawn the new zombie into the array
            zombies[i].spawn(x, y, type, i);

        }
        return zombies;
    }
```

Let's look at all the previous code again, in bite-sized pieces. First, we added the now familiar `include` directives:

```
#include "ZombieArena.h"
#include "Zombie.h"
```

Next comes the function signature. Notice that the function must return a pointer to a `Zombie` object. We will be creating an array of `Zombie` objects. Once we are done creating the horde, we will return the array. When we return the array, we are actually returning the address of the first element of the array. This, as we learned in the previous chapter, is the same thing as a pointer. The signature also shows that we have two parameters. The first, `numZombies`, will be the number of zombies this current horde requires, and the second, `arena`, is an `IntRect` that holds the size of the current arena in which to create this horde.

After the function signature, we declare a pointer to the `Zombie` type called `zombies` and initialize it with the memory address of the first element of an array, which we dynamically allocate on the heap:

```
Zombie* createHorde(int numZombies, IntRect arena)
{
```

```
        Zombie* zombies = new Zombie[numZombies];
```

The next part of the code simply copies the extremities of the arena into `maxY`, `minY`, `maxX`, and `minX`. We subtract 20 pixels from the right and bottom while adding 20 pixels to the top and left. We use these four local variables to help position each of the zombies. We made the 20-pixel adjustments to stop the zombies appearing on top of the walls:

```
int maxY = arena.height - 20;
int minY = arena.top + 20;
int maxX = arena.width - 20;
int minX = arena.left + 20;
```

Now, we enter a `for` loop that will loop through each of the `Zombie` objects in the `zombies` array from zero through to `numZombies`:

```
for (int i = 0; i < numZombies; i++)
```

Inside the `for` loop, the first thing the code does is seed the random number generator and then generate a random number between zero and three. This number is stored in the `side` variable. We will use the `side` variable to decide whether the zombie spawns at the left, top, right, or bottom of the arena. We also declare two `int` variables, `x` and `y`. These two variables will temporarily hold the actual horizontal and vertical coordinates of the current zombie:

```
// Which side should the zombie spawn
srand((int)time(0) * i);
int side = (rand() % 4);
float x, y;
```

Still inside the `for` loop, we have a `switch` block with four `case` statements. Note that the `case` statements are for `0`, `1`, `2`, and `3` and that the argument in the `switch` statement is `side`. Inside each of the `case` blocks, we initialize `x` and `y` with one predetermined value, either `minX`, `maxX`, `minY`, or `maxY`, and one randomly generated value. Look closely at the combinations of each predetermined and random value. You will see that they are appropriate for positioning the current zombie randomly across either the left side, top side, right side, or bottom side. The effect of this will be that each zombie can spawn randomly, anywhere on the outside edge of the arena:

```
switch (side)
{
    case 0:
        // left
        x = minX;
        y = (rand() % maxY) + minY;
        break;
    case 1:
        // right
        x = maxX;
        y = (rand() % maxY) + minY;
        break;
    case 2:
        // top
        x = (rand() % maxX) + minX;
        y = minY;
        break;
    case 3:
        // bottom
        x = (rand() % maxX) + minX;
        y = maxY;
        break;
}
```

Still inside the `for` loop, we seed the random number generator again and generate a random number between 0 and 2. We store this number in the

`type` variable. The `type` variable will determine whether the current zombie will be a Chaser, Bloater, or Crawler.

After the type is determined, we call the `spawn` function on the current `Zombie` object in the `zombies` array. As a reminder, the arguments that are sent into the `spawn` function determine the starting location of the zombie and the type of zombie it will be. The apparently arbitrary `i` is passed in as it is used as a unique seed that randomly varies the speed of a zombie within an appropriate range. This stops our zombies from "bunching up" and becoming a blob rather than a horde:

```
// Bloater, crawler or runner
srand((int)time(0) * i * 2);
int type = (rand() % 3);
// Spawn the new zombie into the array
zombies[i].spawn(x, y, type, i);
```

The `for` loop repeats itself once for each zombie, controlled by the value contained in `numZombies`, and then we return the array. The array, as another reminder, is simply an address of the first element of itself. The array is dynamically allocated on the heap, so it persists after the function returns:

```
return zombies;
```

Now, we can bring our zombies to life.

Bringing the horde to life (or back to life)

We now have a `Zombie` class and a function to make a randomly spawning horde of them. We have the `TextureHolder` singleton as a neat way to hold

just three textures that can be used for dozens or even thousands of zombies. Now, we can add the horde to our game engine in `main`.

Add the following highlighted code to include the `TextureHolder` class. Then, just inside `main`, we will initialize the one and only instance of `TextureHolder`, which can be used from anywhere within our game:

```
#include <SFML/Graphics.hpp>
#include "ZombieArena.h"
#include "Player.h"
#include "TextureHolder.h"
using namespace sf;
int main()
{
 // Here is the instance of TextureHolder
    TextureHolder holder;
// The game will always be in one of four states
enum class State { PAUSED, LEVELING_UP, GAME_OVER, PLAYING };
    // Start with the GAME_OVER state
    State state = State::GAME_OVER;
```

The following few lines of highlighted code declare some control variables for the number of zombies at the start of the wave, the number of zombies still to be killed, and, of course, a pointer to `Zombie` called `zombies` that we initialize to `nullptr`:

```
// Create the background
VertexArray background;
// Load the texture for our background vertex array
Texture textureBackground;
textureBackground.loadFromFile("graphics/background_sheet.png");
// Prepare for a horde of zombies
int numZombies;
int numZombiesAlive;
Zombie* zombies = nullptr;
```

```
// The main game loop
while (window.isOpen())
```

Next, in the `PLAYING` section nested inside the `LEVELING_UP` section, we add code that does the following:

- Initializes `numZombies` to `10`. As the project progresses, this will eventually be dynamic and based on the current wave number.
- Deletes any preexisting allocated memory. Otherwise, each new call to `createHorde` would take up progressively more memory but without freeing up the previous horde's memory.
- Then, we call `createHorde` and assign the returned memory address to `zombies`.
- We also initialize `zombiesAlive` with `numZombies` because we haven't killed any at this point.

Add the following highlighted code, which we have just discussed:

```
if (state == State::PLAYING)
{
    // Prepare the level
// We will modify the next two lines later
    arena.width = 500;
    arena.height = 500;
    arena.left = 0;
    arena.top = 0;
    // Pass the vertex array by reference
// to the createBackground function
int tileSize = createBackground(background, arena);
    // Spawn the player in the middle of the arena
    player.spawn(arena, resolution, tileSize);
 // Create a horde of zombies
    numZombies = 10;
 // Delete the previously allocated memory (if it exists)
 delete[] zombies;
```

```
    zombies = createHorde(numZombies, arena);
    numZombiesAlive = numZombies;
// Reset the clock so there isn't a frame jump
    clock.restart();
}
```

Now, add the following highlighted code to the `ZombieArena.cpp` file:

```
/*
 ****************
 UPDATE THE FRAME
 ****************
 */
if (state == State::PLAYING)
{
    // Update the delta time
    Time dt = clock.restart();
    // Update the total game time
    gameTimeTotal += dt;
    // Make a decimal fraction of 1 from the delta time
float dtAsSeconds = dt.asSeconds();
    // Where is the mouse pointer
    mouseScreenPosition = Mouse::getPosition();
    // Convert mouse position to world coordinates of mainView
    mouseWorldPosition = window.mapPixelToCoords(
        Mouse::getPosition(), mainView);
    // Update the player
    player.update(dtAsSeconds, Mouse::getPosition());
    // Make a note of the players new position
Vector2f playerPosition(player.getCenter());
    // Make the view centre around the player
    mainView.setCenter(player.getCenter());
 // Loop through each Zombie and update them
 for (int i = 0; i < numZombies; i++)
    {
 if (zombies[i].isAlive())
        {
            zombies[i].update(dt.asSeconds(), playerPosition);
        }
```

```
    }
}// End updating the scene
```

All the new preceding code does is loop through the array of zombies, check whether the current zombie is alive, and, if it is, call its `update` function with the necessary arguments.

Add the following code to draw all the zombies:

```
/*
 **************
 Draw the scene
 **************
 */
if (state == State::PLAYING)
{
    window.clear();
    // set the mainView to be displayed in the window
// And draw everything related to it
    window.setView(mainView);
    // Draw the background
    window.draw(background, &textureBackground);
 // Draw the zombies
 for (int i = 0; i < numZombies; i++)
    {
        window.draw(zombies[i].getSprite());
    }
// Draw the player
    window.draw(player.getSprite());
}
```

The preceding code loops through all the zombies and calls the `getSprite` function to allow the `draw` function to do its work. We don't check whether the zombie is alive because even if the zombie is dead, we want to draw the blood splatter.

At the end of the `main` function, we need to make sure to delete our pointer because it is a good practice as well as often being essential. However, technically, this isn't essential because the game is about to exit, and the operating system will reclaim all the memory that's used after the `return 0` statement:

```
    }// End of main game loop
 // Delete the previously allocated memory (if it exists)
 delete[] zombies;
return 0;
}
```

You can run the game and see the zombies spawn around the edge of the arena. They will immediately head straight toward the player at their various speeds. Just for fun, I increased the size of the arena and increased the number of zombies to 1,000, as you can see in the following screenshot:

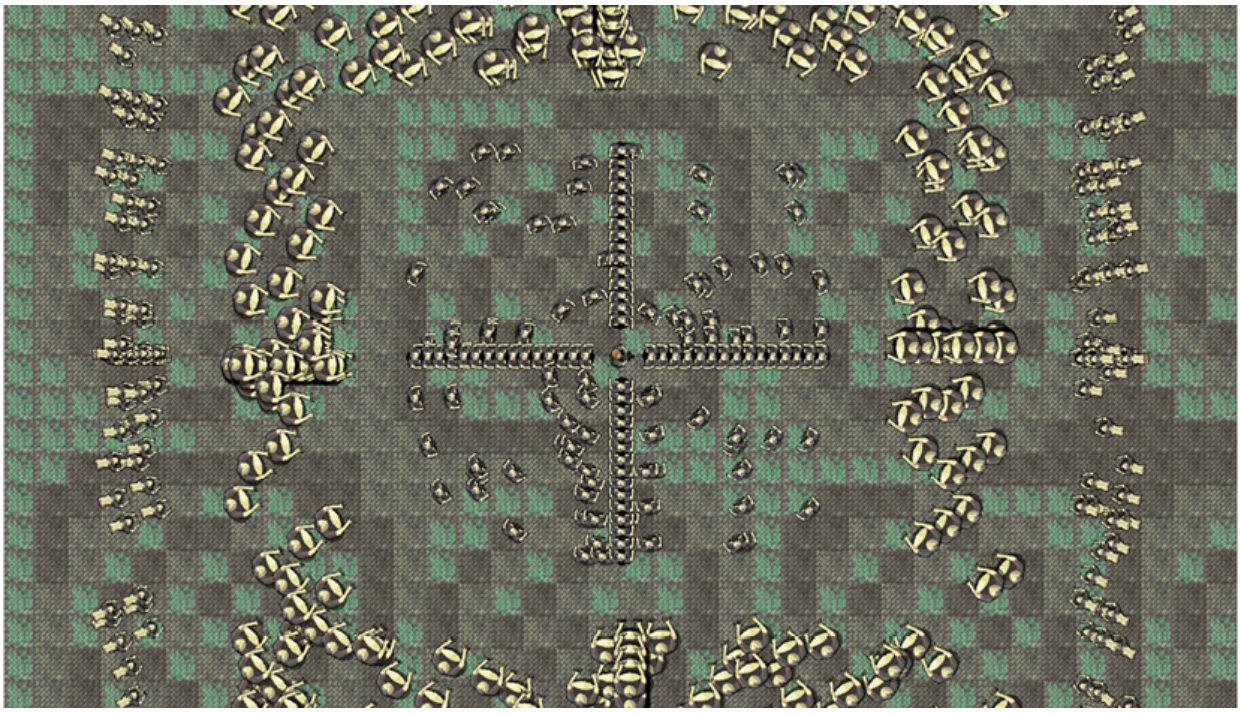


Figure 11.1: Increasing the size of the arena and the number of zombies

This is going to end badly!

Note that you can also pause and resume the onslaught of the horde using the *Enter* key because of the code we wrote in *Chapter 8*, *SFML Views – Starting the Zombie Shooter Game*.

Let's fix the fact that some classes still use a `Texture` instance directly and modify it to use the new `TextureHolder` class.

Using the TextureHolder class for all textures

Since we have our `TextureHolder` class, we might as well be consistent and use it to load all our textures. Let's make some very small alterations to the existing code that loads textures for the background sprite sheet and the player.

Changing the way the background gets its textures

In the `ZombieArena.cpp` file, find the following code:

```
// Load the texture for our background vertex array
Texture textureBackground;
textureBackground.loadFromFile("graphics/background_sheet.png");
```

Delete the code highlighted previously and replace it with the following highlighted code, which uses our new `TextureHolder` class:

```
// Load the texture for our background vertex array
Texture textureBackground = TextureHolder::GetTexture(
 "graphics/background_sheet.png");
```

Let's update the way the `Player` class gets a texture.

Changing the way the Player class gets its texture

In the `Player.cpp` file, inside the constructor, find this code:

```
#include "player.h"
Player::Player()
{
    m_Speed = START_SPEED;
    m_Health = START_HEALTH;
    m_MaxHealth = START_HEALTH;
    // Associate a texture with the sprite
// !!Watch this space!!
    m_Texture.loadFromFile("graphics/player.png");
    m_Sprite.setTexture(m_Texture);
// Set the origin of the sprite to the centre,
// for smooth rotation
    m_Sprite.setOrigin(25, 25);
}
```

Delete the code highlighted previously and replace it with the following highlighted code, which uses our new `TextureHolder` class. In addition, add the `include` directive to add the `TextureHolder` header to the file. The new code is shown highlighted, in context, as follows:

```
#include "player.h"
#include "TextureHolder.h"
Player::Player()
{
```

```
        m_Speed = START_SPEED;
        m_Health = START_HEALTH;
        m_MaxHealth = START_HEALTH;
        // Associate a texture with the sprite
    // !!Watch this space!!
        m_Sprite = Sprite(TextureHolder::GetTexture(
     "graphics/player.png"));
    // Set the origin of the sprite to the centre,
    // for smooth rotation
        m_Sprite.setOrigin(25, 25);
    }
```

From now on, we will use the `TextureHolder` class to load all textures.

Summary

We have built a `TextureHolder` class to contain all the images used as textures by our sprites and coded a `Zombie` class that we can reuse to make as many zombies as we want.

You might have noticed that the zombies don't appear to be very dangerous. They just drift through the player without leaving a scratch. Currently, this is a good thing because the player has no way to defend themself.

In the next chapter, we will make two more classes: one for ammo and health pickups and one for bullets that the player can shoot. After we have done that, we will learn how to detect collisions so that the bullets and zombies do some damage and the pickups can be collected by the player.

Frequently asked questions

Q) Can you remind me about the `new` keyword and memory leaks?

A) When we use memory on the free store using the `new` keyword, it persists even when the function it was created in has returned and all the local variables are gone. When we are done with using memory on the free store, we must release it. So, if we use memory on the free store that we want to persist beyond the life of a function, we must make sure to keep a pointer to it or we will have leaked memory. It would be like putting all our belongings in our house and then forgetting where we live! When we return the `zombies` array from `createHorde`, it is like passing the relay baton (memory address) from `createHorde` to `main`. It's like saying, *OK, here is your horde of zombies; they are your responsibility now*. And, we wouldn't want any leaked zombies running around in our RAM! So, we must remember to call `delete` on pointers to dynamically allocate memory.

12

Collision Detection, Pickups, and Bullets

So far, we have implemented the main visual aspects of our game. We have a controllable character running around in an arena full of zombies that chase them. The problem is that they don't interact with each other. A zombie can wander right through the player without leaving a scratch. We need to detect collisions between the zombies and the player.

If the zombies are going to be able to injure and eventually kill the player, it is only fair that we give the player some bullets for their gun. We will then need to make sure that the bullets can hit and kill the zombies.

At the same time, if we are writing collision detection code for bullets, zombies, and the player, it would be a good time to add a class for health and ammo pickups as well.

Here is what we will do and the order in which we will cover things in this chapter:

- Coding the `Bullet` class
- Making the bullets fly
- Giving the player a crosshair
- Coding a class for pickups
- Using the `Pickup` class

- Detecting collisions

Let's start with the `Bullet` class.

Coding the Bullet class

We will use the SFML `RectangleShape` class to visually represent a bullet. We will code a `Bullet` class that has a `RectangleShape` member, as well as other member data and functions. Then, we will add bullets to our game in a few steps, as follows:

1. First, we will code the `Bullet.h` file. This will reveal all the details of the member data and the prototypes for the functions.
2. Next, we will code the `Bullet.cpp` file, which, of course, will contain the definitions for all the functions of the `Bullet` class. As we step through this, I will explain exactly how an object of the `Bullet` type will work and be controlled.
3. Finally, we will declare a whole array full of bullets in the `main` function. We will also implement a control scheme for shooting, managing the player's remaining ammo, and reloading.

Let's get started with step 1.

Coding the Bullet header file

To make the new header file, right-click **Header Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **Header File (.h)**, and then, in the **Name** field, type `Bullet.h`.

Add the following private member variables, along with the `Bullet` class declaration, to the `Bullet.h` file. We can then run through them and explain what they are for:

```
#pragma once
#include <SFML/Graphics.hpp>
using namespace sf;
class Bullet
{
private:
    // Where is the bullet?
    Vector2f m_Position;
    // What each bullet looks like
    RectangleShape m_BulletShape;
    // Is this bullet currently whizzing through the air
bool m_InFlight = false;
    // How fast does a bullet travel?
float m_BulletSpeed = 1000;
    // What fraction of 1 pixel does the bullet travel,
// Horizontally and vertically each frame?
// These values will be derived from m_BulletSpeed
float m_BulletDistanceX;
    float m_BulletDistanceY;

    // Some boundaries so the bullet doesn't fly forever
float m_MaxX;
    float m_MinX;
    float m_MaxY;
    float m_MinY;
// Public function prototypes go here
};
```

In the previous code, the first member is a `Vector2f` called `m_Position`, which will hold the bullet's location in the game world.

Next, we declare a `RectangleShape` called `m_BulletShape` as we are using a simple non-texture graphic for each bullet, a bit like we did for the time bar

in Timber!

The code then declares a **Boolean**, `m_InFlight`, which will keep track of whether the bullet is currently whizzing through the air or not. This will allow us to decide whether we need to call its `update` function each frame and whether we need to run collision detection checks.

The `float` variable, `m_BulletSpeed`, will (you can probably guess) hold the speed at which the bullet will travel in pixels per second. It is initialized to the value of `1000`, which is a little arbitrary but it works well.

Next, we have two more `float` variables, `m_BulletDistanceX` and `m_BulletDistanceY`. As the calculations to move a bullet are a little more complex than those used to move a zombie or the player, we will benefit from having these two variables on which we will perform calculations. They will be used to decide the horizontal and vertical changes in the bullet's position in each frame.

Finally, we have four more `float` variables (`m_MaxX`, `m_MinX`, `m_MaxY`, and `m_MinY`), which will later be initialized to hold the maximum, minimum, horizontal, and vertical positions for the bullet.

It is likely that the need for some of these variables is not immediately apparent, but it will become clearer when we see each of them in action in the `Bullet.cpp` file.

Now, add all the public function prototypes to the `Bullet.h` file:

```
// Public function prototypes go here
public:
 // The constructor
 Bullet();
 // Stop the bullet
 void stop();
```

```
    // Returns the value of m_InFlight
    bool isInFlight();
    // Launch a new bullet
    void shoot(float startX, float startY,
    float xTarget, float yTarget);
    // Tell the calling code where
    // the bullet is in the world
    FloatRect getPosition();
    // Return the actual shape (for drawing)
    RectangleShape getShape();
    // Update the bullet each frame
    void update(float elapsedTime);
};
```

Let's run through each of the functions in turn, and then we can move on to coding their definitions.

First, we have the `Bullet` function, which is, of course, the constructor. In this function, we will set up each `Bullet` instance, ready for action.

The `stop` function will be called when the bullet has been in action but needs to stop.

The `isInFlight` function returns a Boolean and will be used to test whether a bullet is currently in flight or not.

The `shoot` function's use is given away by its name, but how it will work deserves some discussion. For now, just note that it has four `float` parameters that will be passed in. The four values represent the starting horizontal and vertical position of the bullet (where the player is), as well as the vertical and horizontal target position (where the crosshair is).

The `getPosition` function returns a `FloatRect` that represents the location of the bullet. This function will be used to detect collisions with zombies. You might remember from *Chapter 10*, *Pointers, the Standard Template*

Library, and Texture Management, that zombies also had a `getPosition` function.

Following on, we have the `getShape` function, which returns an object of the `RectangleShape` type. As we have discussed, each bullet is represented visually by a `RectangleShape` object. Therefore, the `getShape` function will be used to grab a copy of the current state of `RectangleShape` in order to draw it.

Finally, and hopefully as expected, there is the `update` function, which has a `float` parameter that represents the fraction of a second that has passed since the last time `update` was called. The `update` method will change the position of the bullet in each frame.

Let's look at and code the function definitions.

Coding the Bullet source file

Now, we can create a new `.cpp` file that will contain the function definitions. Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item**.... In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)**, and then, in the **Name** field, type `Bullet.cpp`. Finally, click the **Add** button. We are now ready to code the class.

Add the following code, which is for the include directives and the constructor. We know it is a constructor because the function has the same name as the class:

```
#include "bullet.h"
// The constructor
Bullet::Bullet()
{
```

```
    m_BulletShape.setSize(sf::Vector2f(2, 2));
}
```

The only thing that the `Bullet` constructor needs to do is set the size of `m_BulletShape`, which is the `RectangleShape` object. The code sets the size to two pixels by two pixels.

Coding the shoot function

Next, we will code the more substantial `shoot` function. Add the following code to the `Bullet.cpp` file and study it, and then we can talk about it:

```
void Bullet::shoot(float startX, float startY,
 float targetX, float targetY)
{
    // Keep track of the bullet
    m_InFlight = true;
    m_Position.x = startX;
    m_Position.y = startY;
    // Calculate the gradient of the flight path
float gradient = (startX - targetX) / (startY - targetY);
    // Any gradient less than 1 needs to be negative
if (gradient < 0)
    {
        gradient *= -1;
    }
    // Calculate the ratio between x and y
float ratioXY = m_BulletSpeed / (1 + gradient);
    // Set the "speed" horizontally and vertically
    m_BulletDistanceY = ratioXY;
    m_BulletDistanceX = ratioXY * gradient;

    // Point the bullet in the right direction
if (targetX < startX)
    {
        m_BulletDistanceX *= -1;
    }
```

```
    if (targetY < startY)
    {
        m_BulletDistanceY *= -1;
    }

    // Set a max range of 1000 pixels
float range = 1000;
    m_MinX = startX - range;
    m_MaxX = startX + range;
    m_MinY = startY - range;
    m_MaxY = startY + range;

    // Position the bullet ready to be drawn
    m_BulletShape.setPosition(m_Position);
}
```

In order to demystify the `shoot` function, we will split it up and talk about the code we have just added in chunks.

First, let's remind ourselves about the signature. The `shoot` function receives the starting and target horizontal and vertical positions of a bullet. The calling code will supply these based on the position of the player sprite and the position of the crosshair. Here it is again:

```
void Bullet::shoot(float startX, float startY,
 float targetX, float targetY)
```

Inside the `shoot` function, we set `m_InFlight` to `true` and position the bullet using the `startX` and `startY` parameters. Here is that piece of code again:

```
// Keep track of the bullet
m_InFlight = true;
```

```
m_Position.x = startX;
m_Position.y = startY;
```

Now, we use a bit of trigonometry to determine the gradient of travel for a bullet. Take a look at the code in question and we will discuss it further and split it up:

```
// Calculate the gradient of the flight path
float gradient = (startX - targetX) / (startY - targetY);
// Any gradient less than zero needs to be negative
if (gradient < 0)
{
    gradient *= -1;
}
// Calculate the ratio between x and y
float ratioXY = m_BulletSpeed / (1 + gradient);
// Set the "speed" horizontally and vertically
m_BulletDistanceY = ratioXY;
m_BulletDistanceX = ratioXY * gradient;
```

The code calculates how a bullet moves toward its target. It adjusts the bullet's path both horizontally and vertically based on the line's slope. This is necessary because if the slope is very steep, the bullet could reach its horizontal destination before it moves enough vertically, or the opposite could happen with less steep angles. Essentially, the code ensures that the bullet travels the correct horizontal and vertical distances at a consistent speed, according to the gradient of the flight path.

Calculating the gradient in the shoot function

Here is the code that calculates the gradient:

```
float gradient = (startX - targetX) / (startY - targetY);
```

This computes the gradient of the flight path using two points, (`startX`, `startY`) and (`targetX`, `targetY`). It subtracts the ending horizontal position from the starting horizontal position. It subtracts the ending vertical position from the starting vertical position and divides the former result by the latter to get a ratio that represents an angle.

Making the gradient positive in the shoot function

Here is the code in question. It's simple but important to our solution:

```
if (gradient < 0)
{
    gradient *= -1;
}
```

This ensures that the gradient is always positive. The negative sign is removed if the gradient is initially negative. This is necessary because the start and target coordinates that are passed in can be negative or positive, and we always want the amount of progression in each frame to be positive. Multiplying by -1 simply changes the negative number to its positive equivalent because a minus multiplied by a minus gives a positive.

Calculating the ratio between X and Y in the shoot function

Look at this next line of code again and then we will further split it up to discuss it:

```
float ratioXY = m_BulletSpeed / (1 + gradient);
```

The `1 + gradient` part adds 1 to the calculated gradient. This is done to prevent division by zero and to ensure that the denominator of the division is not equal to zero.

The part `m_BulletSpeed / (1 + gradient)` calculates the ratio between the horizontal and vertical components of the bullet's movement. The numerator (`m_BulletSpeed`) represents the total speed of the bullet, and the denominator (`1 + gradient`) adjusts this speed based on the slope of the flight path.

If the flight path has a steep upward slope (large positive gradient), the denominator will be larger, resulting in a smaller ratio. This means that more of the bullet's speed is allocated vertically.

If the flight path has a steep downward slope (large negative gradient), the denominator will be smaller, resulting in a larger ratio. This means that more of the bullet's speed is allocated horizontally.

Finally, for this line, `float ratioXY =` stores the result in the variable `ratioXY`. This variable now holds a value that represents the ratio between the horizontal and vertical distances that the bullet should travel based on the calculated gradient and the specified bullet speed.

Finishing the shoot function explanation

The next two lines complete our bullet code:

```
m_BulletDistanceY = ratioXY;
m_BulletDistanceX = ratioXY * gradient;
```

These lines determine how far the bullet should move vertically (`m_BulletDistanceY`) and horizontally (`m_BulletDistanceX`) based on the

previously calculated ratio and the gradient.

Despite all these in-depth calculations, the actual direction of travel will be handled in the `update` function by adding or subtracting the positive values we have just arrived at in this update function.

The next part of the code is much more straightforward. We simply set a maximum horizontal and vertical location that the bullet can reach. We don't want a bullet carrying on forever. In the update function, we will see whether a bullet has passed its maximum or minimum locations:

```
// Set a max range of 1000 pixels in any direction
float range = 1000;
m_MinX = startX - range;
m_MaxX = startX + range;
m_MinY = startY - range;
m_MaxY = startY + range;
```

The following code moves the sprite that represents the bullet to its starting location. We use the `setPosition` function of `Sprite`, as we have often done before:

```
// Position the bullet ready to be drawn
m_BulletShape.setPosition(m_Position);
```

We are now done with the `shoot` function.

More bullet functions

Next, we have four straightforward functions. Let's add the `stop`, `isInFlight`, `getPosition`, and `getShape` functions:

```
void Bullet::stop()
{
    m_InFlight = false;
}
bool Bullet::isInFlight()
{
    return m_InFlight;
}
FloatRect Bullet::getPosition()
{
    return m_BulletShape.getGlobalBounds();
}
RectangleShape Bullet::getShape()
{
    return m_BulletShape;
}
```

The `stop` function simply sets the `m_InFlight` variable to `false`. The `isInFlight` function returns whatever the value of this same variable currently is. So, we can see that `shoot` sets the bullet going, `stop` makes it stop, and `isInFlight` informs us what the current state is.

The `getPosition` function returns a `FloatRect`. We will see how we can use the `FloatRect` from each game object to detect collisions soon.

Finally, for the previous code, `getShape` returns a `RectangleShape` so that we can draw the bullet once each frame.

The Bullet class's update function

The last function we need to implement before we can start using `Bullet` objects is `update`. Add the following code, study it, and then we can talk about it:

```
void Bullet::update(float elapsedTime)
{
    // Update the bullet position variables
    m_Position.x += m_BulletDistanceX * elapsedTime;
    m_Position.y += m_BulletDistanceY * elapsedTime;
    // Move the bullet
    m_BulletShape.setPosition(m_Position);
    // Has the bullet gone out of range?
if (m_Position.x < m_MinX || m_Position.x > m_MaxX ||
        m_Position.y < m_MinY || m_Position.y > m_MaxY)
    {
        m_InFlight = false;
    }
}
```

In the `update` function, we use `m_BulletDistanceX` and `m_BulletDistanceY`, multiplied by the time since the last frame to move the bullet. Remember that the values of the two variables were calculated in the `shoot` function and represent the gradient (ratio to each other) that's required to move the bullet at precisely the correct angle. Then, we use the `setPosition` function to move the `RectangleShape`.

The last thing we do in `update` is a test to see whether the bullet has moved beyond its maximum range. The slightly convoluted `if` statement checks `m_Position.x` and `m_Position.y` against the maximum and minimum values that were calculated in the `shoot` function. These maximum and minimum values are stored in `m_MinX`, `m_MaxX`, `m_MinY`, and `m_MaxY`.

The code checks whether the `m_Position` (`.x` and `.y`) is outside the specified rectangular area defined by `m_MinX`, `m_MaxX`, `m_MinY`, and `m_MaxY`. Remember, the `m_Min…` values define the furthest point the current bullet can travel. If the position is outside this area, the `variable` `m_InFlight` variable is set to `false`, which stops the bullet.

If the test is `true`, then `m_InFlight` is set to `false`.

The `Bullet` class is now done. Next, we will look at how we can shoot some in the `main` function.

Making the bullets fly

In the following sections, we will make the bullets usable with these six steps:

1. Add the necessary include directive for the `Bullet` class.
2. Add some control variables and an array to hold some `Bullet` instances.
3. Handle the player pressing *R* to reload.
4. Handle the player pressing the left mouse button to fire a bullet.
5. Update all bullets that are in flight in each frame.
6. Draw the bullets that are in flight in each frame.

Including the Bullet class

Add the include directive to make the `Bullet` class available:

```
#include <SFML/Graphics.hpp>
#include "ZombieArena.h"
#include "Player.h"
#include "TextureHolder.h"
#include "Bullet.h"
using namespace sf;
```

Let's move on to the next step.

Control variables and the bullet array

Here are some variables to keep track of clip sizes, spare bullets, the remaining bullets in the clip, the current rate of fire (starting at one per second), and the time when the last bullet was fired.

Add the following highlighted code. Then, we can move on and see all these variables in action throughout the rest of this section:

```
// Prepare for a horde of zombies
int numZombies;
int numZombiesAlive;
Zombie* zombies = NULL;
// 100 bullets should do
Bullet bullets[100];
int currentBullet = 0;
int bulletsSpare = 24;
int bulletsInClip = 6;
int clipSize = 6;
float fireRate = 1;
// When was the fire button last pressed?
Time lastPressed;
// The main game loop
while (window.isOpen())
```

Next, let's handle what happens when the player presses the *R* keyboard key, which is used for reloading a clip.

Reloading the gun

Now, we will handle the player input related to shooting bullets. First, we will handle pressing the *R* key to reload the gun. We will do so with an SFML event.

Add the following highlighted code. It is shown with lots of context to make sure the code goes in the right place. Study the code and then we can talk about it:

```
// Handle events
Event event;
while (window.pollEvent(event))
{
    if (event.type == Event::KeyPressed)
    {
        // Pause a game while playing
if (event.key.code == Keyboard::Return &&
            state == State::PLAYING)
        {
            state = State::PAUSED;
        }
        // Restart while paused
else if (event.key.code == Keyboard::Return &&
            state == State::PAUSED)
        {
            state = State::PLAYING;
            // Reset the clock so there
// isn't a frame jump
            clock.restart();
        }
        // Start a new game while in GAME_OVER state
else if (event.key.code == Keyboard::Return &&
            state == State::GAME_OVER)
        {
            state = State::LEVELING_UP;
        }
        if (state == State::PLAYING)
        {
 // Reloading
 if (event.key.code == Keyboard::R)
            {
 if (bulletsSpare >= clipSize)
                {
 // Plenty of bullets. Reload.
                    bulletsInClip = clipSize;
```

```
                    bulletsSpare -= clipSize;
                }
            else if (bulletsSpare > 0)
                {
            // Only few bullets left
                    bulletsInClip = bulletsSpare;
                    bulletsSpare = 0;
                }
            else
                {
            // More here soon?!
                }
            }
        }
    }
}// End event polling
```

The previous code is nested within the event handling part of the game loop (`while(window.pollEvent)`), within the block that only executes when the game is actually being played (`if(state == State::Playing)`). It is obvious that we don't want the player reloading when the game has finished or is paused, and wrapping the new code as we've described achieves this.

In the new code itself, the first thing we do is test for the *R* key being pressed with `if (event.key.code == Keyboard::R)`. Once we have detected that the *R* key was pressed, the remaining code is executed. Here is the structure of the `if`, `else if`, and `else` blocks:

```
if(bulletsSpare >= clipSize)
    ...
else if(bulletsSpare > 0)
    ...
else
    ...
```

The previous structure allows us to handle three possible scenarios, as shown here:

- The player has pressed *R* and they have more bullets spare than the clip can take. In this scenario, the clip is refilled and the number of spare bullets is reduced.
- The player has some spare bullets but not enough to fill the clip completely. In this scenario, the clip is filled with as many spare bullets as the player has and the number of spare bullets is set to zero.
- The player has pressed *R* but they have no spare bullets at all. For this scenario, we don't actually need to alter the variables. However, we will play a sound effect here when we implement the sound in *Chapter 14*, *Sound Effects, File I/O, and Finishing the Game*, so we will leave the empty `else` block ready.

Now, let's shoot a bullet.

Shooting a bullet

Here, we will handle the left mouse button being clicked to fire a bullet. Add the following highlighted code and study it carefully:

```
    if (Keyboard::isKeyPressed(Keyboard::D))
    {
        player.moveRight();
    }
    else
    {
        player.stopRight();
    }
// Fire a bullet
if (Mouse::isButtonPressed(sf::Mouse::Left))
    {
if (gameTimeTotal.asMilliseconds()
```

```
            - lastPressed.asMilliseconds()
            > 1000 / fireRate && bulletsInClip > 0)
        {
 // Pass the centre of the player
 // and the centre of the cross-hair
 // to the shoot function
            bullets[currentBullet].shoot(
                player.getCenter().x, player.getCenter().y,
                mouseWorldPosition.x, mouseWorldPosition.y);
            currentBullet++;
 if (currentBullet > 99)
            {
                currentBullet = 0;
            }
            lastPressed = gameTimeTotal;
            bulletsInClip--;
        }
    }// End fire a bullet
}// End WASD while playing
```

All the previous code is wrapped in an `if` statement that executes whenever the *left mouse button* is pressed, that is, `if (Mouse::isButtonPressed(sf::Mouse::Left))`. Note that the code will execute repeatedly, even if the player just holds down the button. The code we will go through now controls the rate of fire.

In the preceding code, we check whether the total time elapsed in the game (`gameTimeTotal`) minus the time the player last shot a bullet (`lastPressed`) is greater than 1,000, divided by the current rate of fire, and that the player has at least one bullet in the clip. We use 1,000 because this is the number of milliseconds in a second.

If this test is successful, the code that actually fires a bullet is executed. Shooting a bullet is easy because we did all the hard work in the `Bullet` class. We simply call `shoot` on the current bullet from the `bullets` array.

We pass in the player's and the crosshair's current horizontal and vertical locations. The bullet will be configured and set in flight by the code in the `shoot` function of the `Bullet` class.

All we must do is keep track of the array of bullets. We incremented the `currentBullet` variable. Then, we need to check to see whether we fired the last bullet (99) with the `if (currentBullet > 99)` statement. If it was the last bullet, we set `currentBullet` to zero. If it wasn't the last bullet, then the next bullet is ready to go whenever the rate of fire permits it and the player presses the left mouse button.

Finally, in the preceding code, we store the time that the bullet was fired into `lastPressed` and decrement `bulletsInClip`.

Now, we can update every bullet, each frame.

Updating the bullets in each frame

Add the following highlighted code to loop through the `bullets` array, check whether the bullet is in flight, and if it is, call its update function:

```
 // Loop through each Zombie and update them
for (int i = 0; i < numZombies; i++)
    {
        if (zombies[i].isAlive())
        {
            zombies[i].update(dt.asSeconds(), playerPosition);
        }
    }
 // Update any bullets that are in-flight
 for (int i = 0; i < 100; i++)
    {
 if (bullets[i].isInFlight())
        {
            bullets[i].update(dtAsSeconds);
```

```
        }
    }
}// End updating the scene
```

Finally, we will draw all the bullets.

Drawing the bullets in each frame

Add the following highlighted code to loop through the `bullets` array, check whether the bullet is in flight, and if it is, draw it:

```
/*
 **************
 Draw the scene
 **************
 */
if (state == State::PLAYING)
{
    window.clear();
    // set the mainView to be displayed in the window
// And draw everything related to it
    window.setView(mainView);
    // Draw the background
    window.draw(background, &textureBackground);
    // Draw the zombies
for (int i = 0; i < numZombies; i++)
    {
        window.draw(zombies[i].getSprite());
    }
 for (int i = 0; i < 100; i++)
    {
 if (bullets[i].isInFlight())
        {
            window.draw(bullets[i].getShape());
        }
    }
// Draw the player
```

```
        window.draw(player.getSprite());
    }
```

Run the game to try out the bullets. Notice that you can fire six shots before you need to press *R* to reload. The obvious things that are missing are some visual indicators of the number of bullets in the clip and the number of spare bullets. Another problem is that the player can very quickly run out of bullets, especially since the bullets have no stopping power whatsoever. They fly straight through the zombies. Add to this that the player is expected to aim at a mouse pointer instead of a precision crosshair and it is clear that we have work to do.

We will replace the mouse cursor with a **crosshair** next and then spawn some pickups to replenish bullets and health after that. Finally, in this section, we will handle collision detection to make the bullets and the zombies do damage and make the player able to actually get the pickups.

Giving the player a crosshair

Adding a crosshair is easy and only requires one new concept. Add the following highlighted code, and then we can run through it:

```
// 100 bullets should do
Bullet bullets[100];
int currentBullet = 0;
int bulletsSpare = 24;
int bulletsInClip = 6;
int clipSize = 6;
float fireRate = 1;
// When was the fire button last pressed?
Time lastPressed;
// Hide the mouse pointer and replace it with crosshair
window.setMouseCursorVisible(true);
```

```
Sprite spriteCrosshair;
Texture textureCrosshair = TextureHolder::GetTexture("graphics/
spriteCrosshair.setTexture(textureCrosshair);
spriteCrosshair.setOrigin(25, 25);
// The main game loop
while (window.isOpen())
```

First, we call the `setMouseCursorVisible` function on our `window` object. We then load a `Texture`, declare a `Sprite` instance, and initialize it in the usual way. Furthermore, we set the sprite's origin to its center to make it convenient and simpler to make the bullets fly to the middle, as you would expect to happen.

Now, we need to update the crosshair in each frame with the world coordinates of the mouse. Add the following highlighted line of code, which uses the `mouseWorldPosition` vector to set the crosshair's position in each frame:

```
/*
 ****************
 UPDATE THE FRAME
 ****************
 */
if (state == State::PLAYING)
{
    // Update the delta time
    Time dt = clock.restart();
    // Update the total game time
    gameTimeTotal += dt;
    // Make a decimal fraction of 1 from the delta time
float dtAsSeconds = dt.asSeconds();
    // Where is the mouse pointer
    mouseScreenPosition = Mouse::getPosition();
    // Convert mouse position to world coordinates of mainView
    mouseWorldPosition = window.mapPixelToCoords(
        Mouse::getPosition(), mainView);
```

```
 // Set the crosshair to the mouse world location
    spriteCrosshair.setPosition(mouseWorldPosition);
// Update the player
    player.update(dtAsSeconds, Mouse::getPosition());
```

Next, as you have probably come to expect, we can draw the crosshair in each frame. Add the following highlighted line of code in the position shown. This line of code needs no explanation, but its position after all the other game objects is important, so it is drawn on top:

```
/*
 **************
 Draw the scene
 **************
 */
if (state == State::PLAYING)
{
    window.clear();
    // set the mainView to be displayed in the window
// And draw everything related to it
    window.setView(mainView);
    // Draw the background
    window.draw(background, &textureBackground);
    // Draw the zombies
for (int i = 0; i < numZombies; i++)
    {
        window.draw(zombies[i].getSprite());
    }
    for (int i = 0; i < 100; i++)
    {
        if (bullets[i].isInFlight())
        {
            window.draw(bullets[i].getShape());
        }
    }
    // Draw the player
    window.draw(player.getSprite());
 //Draw the crosshair
```

```
    window.draw(spriteCrosshair);
}
```

Now, you can run the game and you will see a cool crosshair instead of a mouse cursor:

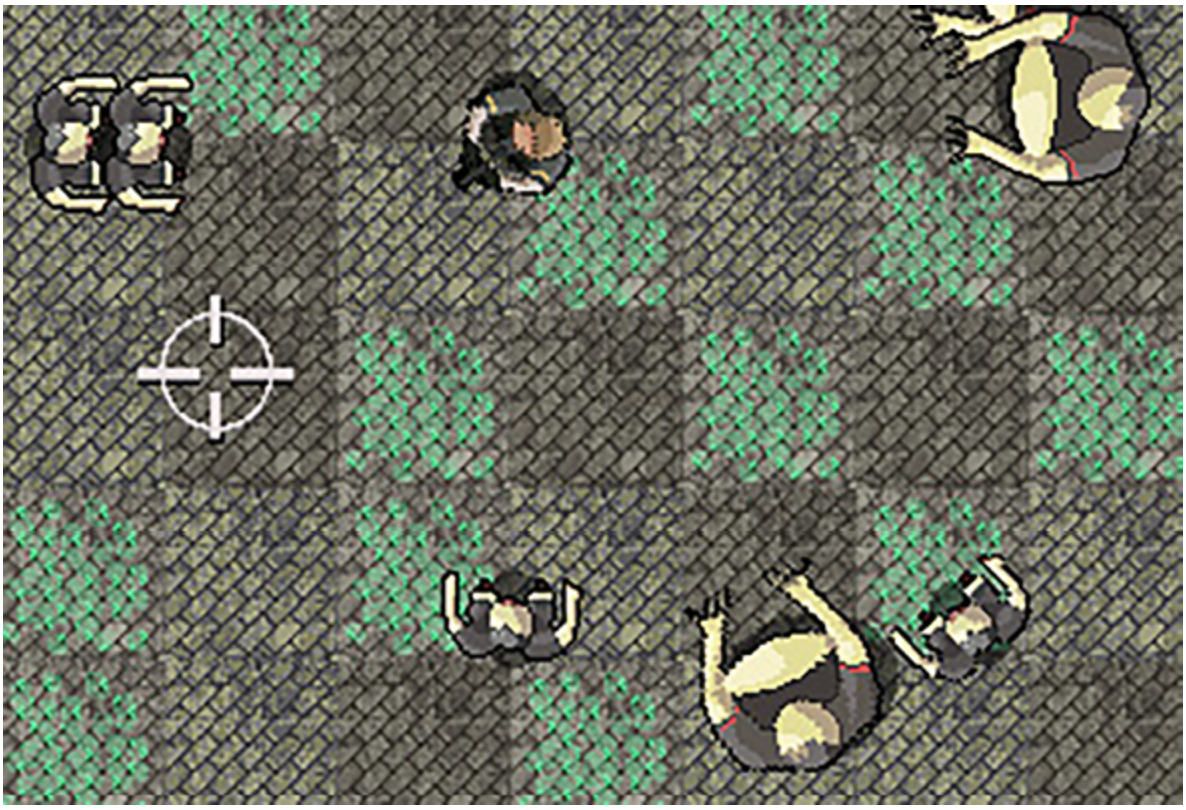


Figure 12.1: A cool crosshair instead of a mouse cursor

Notice how the bullet fires neatly through the center of the crosshair. The way the shooting mechanism works is analogous to allowing the player to choose to shoot from the hip or aim down the sights. If the player keeps the crosshair close to the center, they can fire and turn rapidly, yet must carefully judge the position of distant zombies.

Alternatively, the player can hover their crosshair directly over the head of a distant zombie and score a precise hit; however, they then have much

further to move the crosshair back if a zombie attacks from another direction.

An interesting improvement to the game would be to add a small random amount of inaccuracy to each shot. This inaccuracy could perhaps be mitigated with an upgrade between waves.

Coding a class for pickups

In this section, we will code a `Pickup` class that has a `Sprite` member, as well as other member data and functions. We will add pickups to our game in just a few steps:

1. First, we will code the `Pickup.h` file. This will reveal all the details of the member data and the prototypes for the functions.
2. Then, we will code the `Pickup.cpp` file which, of course, will contain the definitions for all the functions of the `Pickup` class. As we step through this, I will explain exactly how an object of the `Pickup` type will work and be controlled.
3. Finally, we will use the `Pickup` class in the `main` function to spawn them, update them, and draw them.

Let's get started with step 1.

Coding the Pickup header file

To make the new header file, right-click **Header Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **Header File (.h)**, and then, in the **Name** field, type `Pickup.h`.

Add and study the following code to the `Pickup.h` file and then we can go through it:

```
#pragma once
#include <SFML/Graphics.hpp>
using namespace sf;
class Pickup
{
private:
    //Start value for health pickups
const int HEALTH_START_VALUE = 50;
    const int AMMO_START_VALUE = 12;
    const int START_WAIT_TIME = 10;
    const int START_SECONDS_TO_LIVE = 5;

    // The sprite that represents this pickup
    Sprite m_Sprite;
    // The arena it exists in
    IntRect m_Arena;
    // How much is this pickup worth?
int m_Value;

    // What type of pickup is this?
// 1 = health, 2 = ammo
int m_Type;
    // Handle spawning and disappearing
bool m_Spawned;
    float m_SecondsSinceSpawn;
    float m_SecondsSinceDeSpawn;
    float m_SecondsToLive;
    float m_SecondsToWait;
// Public prototypes go here
};
```

The previous code declares all the private variables of the `Pickup` class. Although the names should be quite intuitive, it might not be obvious why many of them are needed at all. Let's go through them, starting from the top:

- `const int HEALTH_START_VALUE = 50`: This constant variable is used to set the starting value of all health pickups. The value will be used to initialize the `m_Value` variable, which will need to be manipulated throughout the game.
- `const int AMMO_START_VALUE = 12`: This constant variable is used to set the starting value of all ammo pickups. The value will be used to initialize the `m_Value` variable, which will need to be manipulated throughout the game.
- `const int START_WAIT_TIME = 10`: This variable determines how long a pickup will wait before it respawns after disappearing. It will be used to initialize the `m_SecondsToWait` variable, which can be manipulated throughout the game.
- `const int START_SECONDS_TO_LIVE = 5`: This variable determines how long a pickup will last between spawning and being de-spawned. Like the previous three constants, it has a non-constant associated with it that can be manipulated throughout the game. The non-constant it uses to initialize is `m_SecondsToLive`.
- `Sprite m_Sprite`: This is the sprite to visually represent the object.
- `IntRect m_Arena`: This will hold the size of the current arena to help the pickup spawn in a sensible position.
- `int m_Value`: How much health or ammo is this pickup worth? This value is used when the player levels up the value of the health or ammo pickup.
- `int m_Type`: This will be either 1 or 2 for health or ammo, respectively. We could have used an enumeration class, but that seemed like overkill for just two options.
- `bool m_Spawned`: Is the pickup currently spawned?

- `float m_SecondsSinceSpawn`: How long has it been since the pickup was spawned?
- `float m_SecondsSinceDeSpawn`: How long has it been since the pickup was de-spawned (disappeared)?
- `float m_SecondsToLive`: How long should this pickup stay spawned before de-spawning?
- `float m_SecondsToWait`: How long should this pickup stay de-spawned before respawning?

Note that most of the complexity of this class is due to the variable spawn time and its upgradable nature. If the pickups just respawned when collected and had a fixed value, this would be a very simple class. We need our pickups to be upgradable so that the player is forced to develop a strategy to progress through the waves.

Next, add the following public function prototypes to the `Pickup.h` file. Be sure to familiarize yourself with the new code so that we can go through it:

```
// Public prototypes go here
public:
 Pickup(int type);
 // Prepare a new pickup
 void setArena(IntRect arena);
 void spawn();
 // Check the position of a pickup
 FloatRect getPosition();
 // Get the sprite for drawing
 Sprite getSprite();
 // Let the pickup update itself each frame
 void update(float elapsedTime);
 // Is this pickup currently spawned?
```

```
    bool isSpawned();
    // Get the goodness from the pickup
    int gotIt();
    // Upgrade the value of each pickup
    void upgrade();
};
```

Let's talk briefly about each of the function definitions:

- The first function is the constructor and is named after the class. Note that it takes a single `int` parameter. This will be used to initialize the type of pickup it will be (health or ammo).
- The `setArena` function receives an `IntRect`. This function will be called for each `Pickup` instance at the start of each wave. The `Pickup` objects will then "know" the areas into which they can spawn.
- The `spawn` function will, of course, handle spawning the pickup.
- The `getPosition` function, just like in the `Player`, `Zombie`, and `Bullet` classes, will return a `FloatRect` instance that represents the current location of the object in the game world.
- The `getSprite` function returns a `Sprite` object that allows the pickup to be drawn once each frame.
- The `update` function receives the time the previous frame took. It uses this value to update its private variables and make decisions about when to spawn and de-spawn.
- The `isSpawned` function returns a Boolean that will let the calling code know whether or not the pickup is currently spawned.
- The `gotIt` function will be called when a collision is detected with the player. The code of the `Pickup` class can then prepare itself for respawning at the appropriate time. Note that it returns an `int` value so

that the calling code knows how much the pickup is "worth" in either health or ammo.

- The `upgrade` function will be called when the player chooses to level up the properties of a pickup during the leveling-up phase of the game.

Now that we have gone through the member variables and function prototypes, it should be quite easy to follow along as we code the function definitions.

Coding the Pickup class function definitions

Now, we can create a new `.cpp` file that will contain the function definitions. Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)**, and then, in the **Name** field, type `Pickup.cpp`. Finally, click the **Add** button. We are now ready to code the class.

Add the following code to the `Pickup.cpp` file. Be sure to review the code so that we can discuss it:

```
#include "Pickup.h"
#include "TextureHolder.h"
Pickup::Pickup(int type): m_Type{type}
{
    // Associate the texture with the sprite
if (m_Type == 1)
    {
        m_Sprite = Sprite(TextureHolder::GetTexture(
            "graphics/health_pickup.png"));
        // How much is pickup worth
        m_Value = HEALTH_START_VALUE;
    }
    else
```

```
        {
            m_Sprite = Sprite(TextureHolder::GetTexture(
                "graphics/ammo_pickup.png"));
            // How much is pickup worth
            m_Value = AMMO_START_VALUE;
        }
        m_Sprite.setOrigin(25, 25);
        m_SecondsToLive = START_SECONDS_TO_LIVE;
        m_SecondsToWait = START_WAIT_TIME;
    }
```

In the previous code, we added the familiar include directives. Then, we added the `Pickup` constructor. We know it is the constructor because it has the same name as the class.

The constructor receives an `int` called `type` and the first thing the code does is assign the value that's received from `type` to `m_Type`. After this, there is an `if else` block that checks whether `m_Type` is equal to `1`. If it is, `m_Sprite` is associated with the health pickup texture and `m_Value` is set to `HEALTH_START_VALUE`.

If `m_Type` is not equal to `1`, the `else` block associates the ammo pickup texture with `m_Sprite` and assigns the value of `AMMO_START_VALUE` to `m_Value`.

After the `if else` block, the code sets the origin of `m_Sprite` to the center using the `setOrigin` function and assigns `START_SECONDS_TO_LIVE` and `START_WAIT_TIME` to `m_SecondsToLive` and `m_SecondsToWait`, respectively.

The constructor has successfully prepared a `Pickup` object that is ready for use.

Now, we will add the `setArena` function. Examine the code as you add it:

```
void Pickup::setArena(IntRect arena)
{
    // Copy the details of the arena to the pickup's m_Arena
    m_Arena.left = arena.left + 50;
    m_Arena.width = arena.width - 50;
    m_Arena.top = arena.top + 50;
    m_Arena.height = arena.height - 50;
    spawn();
}
```

The `setArena` function that we just coded simply copies the values from the passed-in `arena` object but varies the values by `+ 50` on the left and top and `- 50` on the right and bottom. The `Pickup` object is now aware of the area in which it can spawn. The `setArena` function then calls its own `spawn` function to make the final preparations for being drawn and updated each frame.

The `spawn` function is next. Add the following code after the `setArena` function:

```
void Pickup::spawn()
{
    // Spawn at a random location
srand((int)time(0) / m_Type);
    int x = (rand() % m_Arena.width);
    srand((int)time(0) * m_Type);
    int y = (rand() % m_Arena.height);
    m_SecondsSinceSpawn = 0;
    m_Spawned = true;
    m_Sprite.setPosition(x, y);
}
```

The `spawn` function does everything necessary to prepare the pickup. First, it seeds the random number generator and gets a random number for both the horizontal and vertical position of the object. Notice that it uses the

`m_Arena.width` and `m_Arena.height` variables as the ranges for the possible horizontal and vertical positions.

The `m_SecondsSinceSpawn` variable is set to zero so that the length of time that's allowed before it is de-spawned is reset. The `m_Spawned` variable is set to `true` so that, when we call `isSpawned`, from `main`, we will get a positive response. Finally, `m_Sprite` is moved into position with `setPosition`, ready to be drawn to the screen.

In the following block of code, we have three simple getter functions. The `getPosition` function returns a `FloatRect` of the current position of `m_Sprite`, `getSprite` returns a copy of `m_Sprite` itself, and `isSpawned` returns `true` or `false`, depending on whether the object is currently spawned.

Add and examine the code we have just discussed:

```
FloatRect Pickup::getPosition()
{
    return m_Sprite.getGlobalBounds();
}
Sprite Pickup::getSprite()
{
    return m_Sprite;
}
bool Pickup::isSpawned()
{
    return m_Spawned;
}
```

Next, we will code the `gotIt` function. This function will be called from `main` when the player touches/collides with (gets) the pickup. Add the `gotIt` function after the `isSpawned` function:

```
int Pickup::gotIt()
{
    m_Spawned = false;
    m_SecondsSinceDeSpawn = 0;
    return m_Value;
}
```

The `gotIt` function sets `m_Spawned` to `false` so that we know not to draw and check for collisions anymore. `m_SecondsSinceDespawn` is set to zero so that the countdown to spawning begins again from the start. `m_Value` is then returned to the calling code so that the calling code can handle adding extra ammunition or health, as appropriate.

Following this, we need to code the `update` function, which ties together many of the variables and functions we have seen so far. Add and familiarize yourself with the `update` function, and then we can talk about it:

```
void Pickup::update(float elapsedTime)
{
    if (m_Spawned)
    {
        m_SecondsSinceSpawn += elapsedTime;
    }
    else
    {
        m_SecondsSinceDeSpawn += elapsedTime;
    }
    // Do we need to hide a pickup?
if (m_SecondsSinceSpawn > m_SecondsToLive && m_Spawned)
    {
        // Remove the pickup and put it somewhere else
        m_Spawned = false;
        m_SecondsSinceDeSpawn = 0;
    }
    // Do we need to spawn a pickup
if (m_SecondsSinceDeSpawn > m_SecondsToWait && !m_Spawned)
    {
```

```
            // spawn the pickup and reset the timer
    spawn();
        }
    }
```

The `update` function is divided into four blocks that are considered for execution in each frame:

- An `if` block that executes if `m_Spawned` is true: `if (m_Spawned)`. This block of code adds the time from this frame to `m_SecondsSinceSpawned`, which keeps track of how long the pickup has been spawned.
- A corresponding `else` block that executes if `m_Spawned` is `false`. This block adds the time this frame took to `m_SecondsSinceDeSpawn`, which keeps track of how long the pickup has waited since it was last de-spawned (hidden).
- Another `if` block that executes when the pickup has been spawned for longer than it should have been: `if (m_SecondsSinceSpawn > m_SecondsToLive && m_Spawned)`. This block sets `m_Spawned` to `false` and resets `m_SecondsSinceDeSpawn` to zero. Now, block 2 will execute until it is time to spawn it again.
- A final `if` block that executes when the time to wait since de-spawning has exceeded the necessary wait time, and the pickup is not currently spawned: `if (m_SecondsSinceDeSpawn > m_SecondsToWait && !m_Spawned)`. When this block is executed, it is time to spawn the pickup again, and the `spawn` function is called.

These four tests are what control the hiding and showing of a pickup.

Finally, add the definition for the `upgrade` function:

```
void Pickup::upgrade()
{
    if (m_Type == 1)
    {
        m_Value += (HEALTH_START_VALUE * .5);
    }
    else
    {
        m_Value += (AMMO_START_VALUE * .5);
    }
    // Make them more frequent and last longer
    m_SecondsToLive += (START_SECONDS_TO_LIVE / 10);
    m_SecondsToWait -= (START_WAIT_TIME / 10);
}
```

The `upgrade` function tests for the type of pickup, either health or ammo, and then adds 50% of the (appropriate) starting value to `m_Value`. The next two lines after the `if else` blocks increase the amount of time the pickup will remain spawned and decrease the amount of time the player must wait between spawns.

This function is called when the player chooses to level up the pickups during the `LEVELING_UP` state.

Our `Pickup` class is ready for use.

Using the Pickup class

After all that hard work implementing the `Pickup` class, we can now go ahead and write code in the game engine to put some pickups into the game.

The first thing we will do is add an `include` directive to the `ZombieArena.cpp` file:

```
#include <SFML/Graphics.hpp>
#include "ZombieArena.h"
#include "Player.h"
#include "TextureHolder.h"
#include "Bullet.h"
#include "Pickup.h"
using namespace sf;
```

In the following code, we are adding two `Pickup` instances: one called `healthPickup` and another called `ammoPickup`. We pass the values 1 and 2, respectively, into the constructor so that they are initialized to the correct type of pickup. Add the following highlighted code, which we have just discussed:

```
// Hide the mouse pointer and replace it with crosshair
window.setMouseCursorVisible(true);
Sprite spriteCrosshair;
Texture textureCrosshair = TextureHolder::GetTexture(
        "graphics/crosshair.png");
spriteCrosshair.setTexture(textureCrosshair);
spriteCrosshair.setOrigin(25, 25);
// Create a couple of pickups
Pickup healthPickup(1);
Pickup ammoPickup(2);
// The main game loop
while (window.isOpen())
```

In the `LEVELING_UP` state of the keyboard handling, add the following highlighted lines within the nested `PLAYING` code block:

```
if (state == State::PLAYING)
{
    // Prepare the level
// We will modify the next two lines later
    arena.width = 500;
    arena.height = 500;
```

```
        arena.left = 0;
        arena.top = 0;
        // Pass the vertex array by reference
    // to the createBackground function
    int tileSize = createBackground(background, arena);
        // Spawn the player in the middle of the arena
        player.spawn(arena, resolution, tileSize);
     // Configure the pick-ups
        healthPickup.setArena(arena);
        ammoPickup.setArena(arena);
    // Create a horde of zombies
        numZombies = 10;
        // Delete the previously allocated memory (if it exists)
    delete[] zombies;
        zombies = createHorde(numZombies, arena);
        numZombiesAlive = numZombies;
        // Reset the clock so there isn't a frame jump
        clock.restart();
    }
```

The preceding code simply passes `arena` into the `setArena` function of each pickup. The pickups now know where they can spawn. This code executes for each new wave, so, as the arena's size grows, the `Pickup` objects will get updated.

The following code simply calls the `update` function for each `Pickup` object on each frame:

```
    // Loop through each Zombie and update them
    for (int i = 0; i < numZombies; i++)
        {
            if (zombies[i].isAlive())
            {
                zombies[i].update(dt.asSeconds(), playerPosition);
            }
        }
        // Update any bullets that are in-flight
    for (int i = 0; i < 100; i++)
```

```
        {
            if (bullets[i].isInFlight())
            {
                bullets[i].update(dtAsSeconds);
            }
        }
    // Update the pickups
        healthPickup.update(dtAsSeconds);
        ammoPickup.update(dtAsSeconds);
    }// End updating the scene
```

The following code in the draw part of the game loop checks whether the pickup is currently spawned and, if it is, draws it. Let's add it:

```
    // Draw the player
        window.draw(player.getSprite());
    // Draw the pick-ups, if currently spawned
    if (ammoPickup.isSpawned())
        {
            window.draw(ammoPickup.getSprite());
        }

    if (healthPickup.isSpawned())
        {
            window.draw(healthPickup.getSprite());
        }
     //Draw the crosshair
        window.draw(spriteCrosshair);
    }
```

Now, you can run the game and see the pickups spawn and de-spawn. You can't, however, actually pick them up yet:

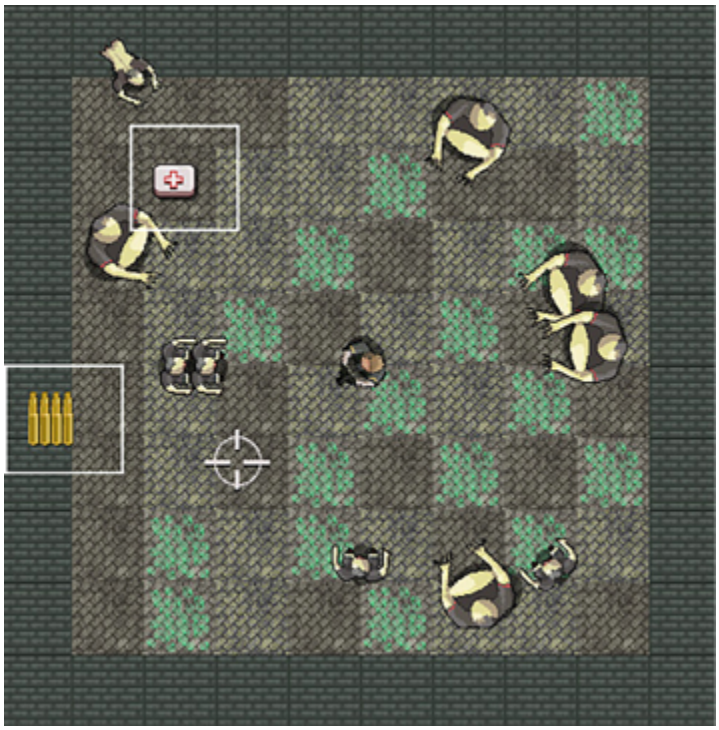


Figure 12.2: Pickups spawn and de-spawn

Now that we have all the objects in our game, it is a good time to make them interact (collide) with each other.

Detecting collisions

We just need to know when certain objects from our game touch certain other objects. We can then respond to that event in an appropriate manner. In our classes, we have already added functions that will be called when our objects collide. They are as follows:

- The `Player` class has a `hit` function. We will call it when a zombie collides with the player.
- The `Zombie` class has a `hit` function. We will call it when a bullet collides with a zombie.

- The `Pickup` class has a `gotIt` function. We will call it when the player collides with a pickup.

If necessary, look back to refresh your memory regarding how each of those functions works. All we need to do now is detect the collisions and call the appropriate functions.

We will use **rectangle intersection** to detect collisions. This type of collision detection is straightforward (especially with SFML). We will use the same technique that we used in the Pong game. The following image shows how a rectangle can reasonably accurately represent the zombies and the player:

Figure 12.3 Rectangle representing the zombies and the player

We will deal with this in three sections of code that will all follow on from one another. They will all go at the end of the update part of our game engine.

We need to know the answers to the following three questions for each frame:

1. Has a zombie been shot?
2. Has the player been touched by a zombie?
3. Has the player touched a pickup?

First, let's add a couple more variables for `score` and `hiscore`. We can then change them when a zombie is killed. Add the following code:

```
// Create a couple of pickups
Pickup healthPickup(1);
Pickup ammoPickup(2);
// About the game
int score = 0;
int hiScore = 0;
// The main game loop
while (window.isOpen())
```

Now, let's start by detecting whether a zombie is colliding with a bullet.

Has a zombie been shot?

The following code might look complicated but, when we step through it, we will see it is nothing we haven't seen before. Add the following code just after the call to update the pickups for each frame. Then, we can go through it:

```
// Update the pickups
healthPickup.update(dtAsSeconds);
ammoPickup.update(dtAsSeconds);
// Collision detection
// Have any zombies been shot?
for (int i = 0; i < 100; i++)
{
 for (int j = 0; j < numZombies; j++)
    {
 if (bullets[i].isInFlight() &&
```

```
            zombies[j].isAlive())
        {
 if (bullets[i].getPosition().intersects
                (zombies[j].getPosition()))
            {
 // Stop the bullet
                bullets[i].stop();
 // Register the hit and see if it was a kill
 if (zombies[j].hit())
                {
 // Not just a hit but a kill too
                    score += 10;
 if (score >= hiScore)
                    {
                        hiScore = score;
                    }
                    numZombiesAlive--;
 // When all the zombies are dead (again)
 if (numZombiesAlive == 0) {
                        state = State::LEVELING_UP;
                    }
                }

            }
        }
    }
}// End zombie being shot
```

In the next section, we will see all the zombie and bullet collision detection code again. We will do so a bit at a time so that we can discuss it. First, notice the structure of the nested `for` loops in the preceding code (with some code stripped out), as shown again here:

```
// Collision detection
// Have any zombies been shot?
for (int i = 0; i < 100; i++)
{
    for (int j = 0; j < numZombies; j++)
```

```
        {
            ...
            ...
            ...
        }
    }
```

The code loops through every bullet (0 to 99) for every zombie (0 to less than `numZombies`.).

Within the nested `for` loops, we do the following:

1. We check whether the current bullet is in flight and the current zombie is still alive with the following code:

```
if (bullets[i].isInFlight() && zombies[j].isAlive())
```

2. Provided the zombie is alive and the bullet is in flight, we test for a rectangle intersection with the following code:

```
if (bullets[i].getPosition().intersects(zombies[j].ge
```

3. If the current bullet and zombie have collided, then we take a number of steps, as detailed next.Stop the bullet with the following code:

```
// Stop the bullet
bullets[i].stop();
```

4. Register a hit with the current zombie by calling its `hit` function. Note that the `hit` function returns a Boolean that lets the calling code know whether the zombie is dead yet. This is shown in the following line of code:

```
// Register the hit and see if it was a kill
if (zombies[j].hit()) {
```

5. Inside this `if` block, which detects when the zombie is dead and hasn't just wounded us, do the following:
    - Add 10 to `score`.
    - Change `hiScore` if the score the player has achieved or exceeded (beaten) `score`.
    - Reduce `numZombiesAlive` by one.
    - Check whether all the zombies are dead with `(numZombiesAlive == 0)` and, if so, change `state` to `LEVELING_UP`.

Here is the block of code inside `if(zombies[j].hit())` that we have just discussed:

```
// Not just a hit but a kill too
score += 10;
if (score >= hiScore)
{
    hiScore = score;
}
numZombiesAlive--;
// When all the zombies are dead (again)
if (numZombiesAlive == 0)
{
    state = State::LEVELING_UP;
}
```

That's the zombies and the bullets taken care of. You can now run the game and see the blood. Of course, you won't see the score until we implement the HUD in the next chapter.

Has the player been touched by a zombie?

This code is much shorter and simpler than the zombie and bullet collision detection code. Add the following highlighted code just after the previous code we wrote:

```
}// End zombie being shot
// Have any zombies touched the player
for (int i = 0; i < numZombies; i++)
{
 if (player.getPosition().intersects
        (zombies[i].getPosition()) && zombies[i].isAlive())
    {
 if (player.hit(gameTimeTotal))
        {
 // More here later
        }
 if (player.getHealth() <= 0)
        {
            state = State::GAME_OVER;
        }
    }
}// End player touched
```

Here, we detect whether a zombie has collided with the player by using a `for` loop to go through all the zombies. For each zombie that is alive, the code uses the `intersects` function to test for a collision with the player. When a collision has occurred, we call `player.hit`. Then, we check whether the player is dead by calling `player.getHealth`. If the player's health is equal to or less than zero, we change `state` to `GAME_OVER`.

You can run the game and collisions will be detected. However, as there is no HUD or sound effects yet, it is not clear that this is happening. In

addition, we need to do some more work resetting the game when the player has died and a new game is starting.

So, although the game runs, the results are not especially satisfying right now. We will improve this over the next two chapters.

Has the player touched a pickup?

The collision detection code between the player and each of the two pickups is shown here. Add the following highlighted code just after the previous code that we added:

```
    }// End player touched
 // Has the player touched health pickup
 if (player.getPosition().intersects
         (healthPickup.getPosition()) && healthPickup.isSpawned(
     {
         player.increaseHealthLevel(healthPickup.gotIt());

     }
 // Has the player touched ammo pickup
 if (player.getPosition().intersects
         (ammoPickup.getPosition()) && ammoPickup.isSpawned())
     {
         bulletsSpare += ammoPickup.gotIt();

     }
}// End updating the scene
```

The preceding code uses two simple `if` statements to see whether either `healthPickup` or `ammoPickup` has been touched by the player.

If a health pickup has been collected, then the `player.increaseHealthLevel` function uses the value returned from the `healthPickup.gotIt` function to

increase the player's health.

If an ammo pickup has been collected, then `bulletsSpare` is increased by the value that's returned from `ammoPickup.gotIt`.

You can now run the game, kill zombies, and collect pickups! Note that, when your health equals zero, the game will enter the `GAME_OVER` state and pause. To restart it, you will need to press *Enter*, followed by a number between *1* and *6*. When we implement the HUD, the home screen, and the leveling-up screen, these steps will be intuitive and straightforward for the player. We will do so in the next chapter.

Summary

This was a busy chapter, but we achieved a lot. Not only did we add bullets and pickups to the game through two new classes but we also made all the objects interact as they should by detecting when they collide with each other.

Despite these achievements, we need to do more work to set up each new game and to give the player feedback through a HUD. In the next chapter, we will build the HUD.

Frequently asked questions

Here is a question that might be on your mind:

Q1) Are there any better ways of doing collision detection?

A) Yes. There are lots more ways to do collision detection, including but not limited to the following:

- You can divide objects up into multiple rectangles that fit the shape of the sprite better. It is perfectly manageable for C++ to check on thousands of rectangles in each frame. This is especially the case when you use techniques such as neighbor checking to reduce the number of tests that are necessary for each frame.
- For circular objects, you can use the radius overlap method.
- For irregular polygons, you can use the passing number algorithm.

You can review all of these techniques, if you wish, by taking a look at the following links:

- Neighbor checking: http://gamecodeschool.com/essentials/collision-detection-neighbor-checking/
- Radius overlap method: http://gamecodeschool.com/essentials/collision-detection-radius-overlap/
- Crossing number algorithm: http://gamecodeschool.com/essentials/collision-detection-crossing-number/

13

Layering Views and Implementing the HUD

In this chapter, we will get to see the real value of SFML Views. We will add a selection of SFML `Text` objects and manipulate them as we did before in the Timber!!! project and the Pong project. What's new is that we will draw the **HUD** using a second `View` instance. This way, the HUD will stay neatly positioned over the top of the main game action, regardless of what the background, player, zombies, and other game objects are doing.

Here is what we will do in this chapter:

- Adding all the `Text` and HUD objects
- Updating the HUD
- Drawing the HUD, home, and level-up screens

Adding all the Text and HUD objects

We will be manipulating a few strings in this chapter. We are doing this so that we can format the HUD and the level-up screen with the necessary text.

Add the extra `include` directive to the `ZombieArena.cpp` file as highlighted in the following code so that we can make some `sstream` objects to achieve this:

```
#include <sstream>
#include <SFML/Graphics.hpp>
#include "ZombieArena.h"
#include "Player.h"
#include "TextureHolder.h"
#include "Bullet.h"
#include "Pickup.h"
using namespace sf;
```

Next, add this rather lengthy, but easily explainable, piece of code. To help identify where you should add the code, the new code is highlighted, and the existing code is not:

```
int score = 0;
int hiScore = 0;
// For the home/game over screen
Sprite spriteGameOver;
Texture textureGameOver = TextureHolder::GetTexture("graphics/ba
spriteGameOver.setTexture(textureGameOver);
spriteGameOver.setPosition(0, 0);
// Create a view for the HUD
View hudView(sf::FloatRect(0, 0, 1920,1080));
// Create a sprite for the ammo icon
Sprite spriteAmmoIcon;
Texture textureAmmoIcon = TextureHolder::GetTexture(
 "graphics/ammo_icon.png");
spriteAmmoIcon.setTexture(textureAmmoIcon);
spriteAmmoIcon.setPosition(20, 980);
// Load the font
Font font;
font.loadFromFile("fonts/zombiecontrol.ttf");
// Paused
Text pausedText;
pausedText.setFont(font);
pausedText.setCharacterSize(155);
pausedText.setFillColor(Color::White);
pausedText.setPosition(400, 400);
pausedText.setString("Press Enter \nto continue");
// Game Over
```

```
Text gameOverText;
gameOverText.setFont(font);
gameOverText.setCharacterSize(125);
gameOverText.setFillColor(Color::White);
gameOverText.setPosition(250, 850);
gameOverText.setString("Press Enter to play");
// LEVELING up
Text levelUpText;
levelUpText.setFont(font);
levelUpText.setCharacterSize(80);
levelUpText.setFillColor(Color::White);
levelUpText.setPosition(150, 250);
std::stringstream levelUpStream;
levelUpStream <<
 "1- Increased rate of fire" <<
 "\n2- Increased clip size(next reload)" <<
 "\n3- Increased max health" <<
 "\n4- Increased run speed" <<
 "\n5- More and better health pickups" <<
 "\n6- More and better ammo pickups";
levelUpText.setString(levelUpStream.str());
// Ammo
Text ammoText;
ammoText.setFont(font);
ammoText.setCharacterSize(55);
ammoText.setFillColor(Color::White);
ammoText.setPosition(200, 980);
// Score
Text scoreText;
scoreText.setFont(font);
scoreText.setCharacterSize(55);
scoreText.setFillColor(Color::White);
scoreText.setPosition(20, 0);
// Hi Score
Text hiScoreText;
hiScoreText.setFont(font);
hiScoreText.setCharacterSize(55);
hiScoreText.setFillColor(Color::White);
hiScoreText.setPosition(1400, 0);
std::stringstream s;
s << "Hi Score:" << hiScore;
hiScoreText.setString(s.str());
// Zombies remaining
```

```
Text zombiesRemainingText;
zombiesRemainingText.setFont(font);
zombiesRemainingText.setCharacterSize(55);
zombiesRemainingText.setFillColor(Color::White);
zombiesRemainingText.setPosition(1500, 980);
zombiesRemainingText.setString("Zombies: 100");
// Wave number
int wave = 0;
Text waveNumberText;
waveNumberText.setFont(font);
waveNumberText.setCharacterSize(55);
waveNumberText.setFillColor(Color::White);
waveNumberText.setPosition(1250, 980);
waveNumberText.setString("Wave: 0");
// Health bar
RectangleShape healthBar;
healthBar.setFillColor(Color::Red);
healthBar.setPosition(450, 980);
// The main game loop
while (window.isOpen())
```

The previous code is very simple and nothing new. It basically creates a whole bunch of SFML `Text` objects. It assigns their colors and sizes and then formats their positions using functions we have seen before.

The most important thing to note is that we create another `View` object called `hudView` and initialize it to fit the resolution of the screen.

As we have seen, the main `View` object scrolls around as it follows the player. In contrast, we will never move `hudView`. The result of this is that if we switch to this view before we draw the elements of the HUD, we will create the effect of allowing the game world to scroll by underneath while the player's HUD remains stationary.

> As an analogy, you can think of laying a transparent sheet of plastic with some writing on it over a TV screen. The TV

will carry on as normal with moving pictures, and the text on the plastic sheet will stay in the same place, regardless of what goes on underneath it. We will take this concept a step further in the next project when we create a platform game with moving views of the game world.

The next thing to notice, however, is that the hi-score is not set in any meaningful way. We will need to wait until the next chapter, when we investigate file I/O, to save and retrieve the high score.

Another point worth noting is that we declare and initialize a `RectangleShape` called `healthBar`, which will be a visual representation of the player's remaining health. This will work in almost the same way that the time-bar worked in the Timber!!! project, except it will represent health instead of time.

In the previous code, there is a new `Sprite` instance called `ammoIcon` that gives context to the bullet and clip statistics that we will draw next to it, at the bottom left of the screen.

Although there is nothing new or technical about the large amount of code that we just added, be sure to familiarize yourself with the details – especially the variable names – to make the rest of this chapter easier to follow.

Now let's get introduced to updating the HUD variables.

Updating the HUD

As you might expect, we will update the HUD variables in the updated section of our code. However, we will not do so at every frame. The reason

for this is that it is unnecessary, and it also slows our game loop down.

As an example, consider the scenario when the player kills a zombie and gets some more points. It doesn't matter whether the `Text` object that holds the score is updated in one-thousandth, one-hundredth, or even one-tenth of a second. The player will discern no difference. This means there is no point rebuilding strings that we set for the `Text` objects every frame.

Therefore, we can time when and how often we update the HUD. Add the following highlighted variables:

```
// Health bar
RectangleShape healthBar;
healthBar.setFillColor(Color::Red);
healthBar.setPosition(450, 980);
// When did we last update the HUD?
int framesSinceLastHUDUpdate = 0;
// How often (in frames) should we update the HUD
int fpsMeasurementFrameInterval = 1000;
// The main game loop
while (window.isOpen())
```

In the previous code, we have variables to track how many frames it has been since the last time the HUD was updated, and the interval, measured in frames, we would like to wait between HUD updates.

Now, we can use these new variables and update the HUD for each frame. We won't see all the HUD elements change, however, until we begin to manipulate the final variables, such as `wave`, in the next chapter.

Add the following highlighted code in the updated section of the game loop, as follows:

```
 // Has the player touched ammo pickup
if (player.getPosition().intersects
```

```
            (ammoPickup.getPosition()) && ammoPickup.isSpawned())
        {
            bulletsSpare += ammoPickup.gotIt();

        }
    // size up the health bar
        healthBar.setSize(Vector2f(player.getHealth() * 3, 50));
    // Increment the number of frames since the previous update
        framesSinceLastHUDUpdate++;
    // re-calculate every fpsMeasurementFrameInterval frames
    if (framesSinceLastHUDUpdate > fpsMeasurementFrameInterval)
        {
    // Update game HUD text
            std::stringstream ssAmmo;
            std::stringstream ssScore;
            std::stringstream ssHiScore;
            std::stringstream ssWave;
            std::stringstream ssZombiesAlive;
    // Update the ammo text
            ssAmmo << bulletsInClip << "/" << bulletsSpare;
            ammoText.setString(ssAmmo.str());
    // Update the score text
            ssScore << "Score:" << score;
            scoreText.setString(ssScore.str());
    // Update the high score text
            ssHiScore << "Hi Score:" << hiScore;
            hiScoreText.setString(ssHiScore.str());
    // Update the wave
            ssWave << "Wave:" << wave;
            waveNumberText.setString(ssWave.str());
    // Update the high score text
            ssZombiesAlive << "Zombies:" << numZombiesAlive;
            zombiesRemainingText.setString(ssZombiesAlive.str());
            framesSinceLastHUDUpdate = 0;
        }// End HUD update
    }// End updating the scene
```

In the new code, we update the size of the `healthBar` sprite and then increment the `framesSinceLastHUDUpdate` variable.

Next, we start an `if` block that tests whether `framesSinceLastHUDUpdate` is greater than our preferred interval, which is stored in `fpsMeasurementFrameInterval`.

Inside this `if` block is where all the action takes place. First, we declare a `stringstream` object for each string that we need to set to a `Text` object.

Then, we use each of those `stringstream` objects in turn and use the `setString` function to set the result to the appropriate `Text` object.

Finally, before the `if` block is exited, `framesSinceLastHUDUpdate` is set back to `0` so that the count can begin again.

Now, when we redraw the scene, the new values will appear in the player's HUD.

Drawing the HUD, home, and level-up screens

All the code in the following three code blocks goes in the drawing phase of our game loop. All we need to do is draw the appropriate `Text` objects during the appropriate states, in the draw section of the main game loop.

In the `PLAYING` state, add the following highlighted code:

```
    //Draw the crosshair
        window.draw(spriteCrosshair);
        // Draw the player
        window.draw(player.getSprite());
    // Switch to the HUD view
        window.setView(hudView);
    // Draw all the HUD elements
        window.draw(spriteAmmoIcon);
        window.draw(ammoText);
```

```
        window.draw(scoreText);
        window.draw(hiScoreText);
        window.draw(healthBar);
        window.draw(waveNumberText);
        window.draw(zombiesRemainingText);
    }
    if (state == State::LEVELING_UP)
    {
    }
```

The vital thing to notice in the preceding block of code is that we switch views to the HUD view. This causes everything to be drawn at the precise screen positions we gave each of the elements of the HUD. They will never move because we never change the HUD view.

In the `LEVELING_UP` state, add the following highlighted code:

```
    if (state == State::LEVELING_UP)
    {
        window.draw(spriteGameOver);
        window.draw(levelUpText);
    }
```

In the `PAUSED` state, add the following highlighted code:

```
    if (state == State::PAUSED)
    {
        window.draw(pausedText);
    }
```

In the `GAME_OVER` state, add the following highlighted code:

```
    if (state == State::GAME_OVER)
    {
        window.draw(spriteGameOver);
        window.draw(gameOverText);
```

```
    window.draw(scoreText);
    window.draw(hiScoreText);
}
```

Now, we can run the game and see our HUD update during gameplay:

Figure 13.1: HUD update during gameplay

The following screenshot shows the high score and score on the home/game over screen:

Figure 13.2: High score and score on the home/game over screen

Next, we see text that tells the player what their level-up options are, although these options don't do anything yet:

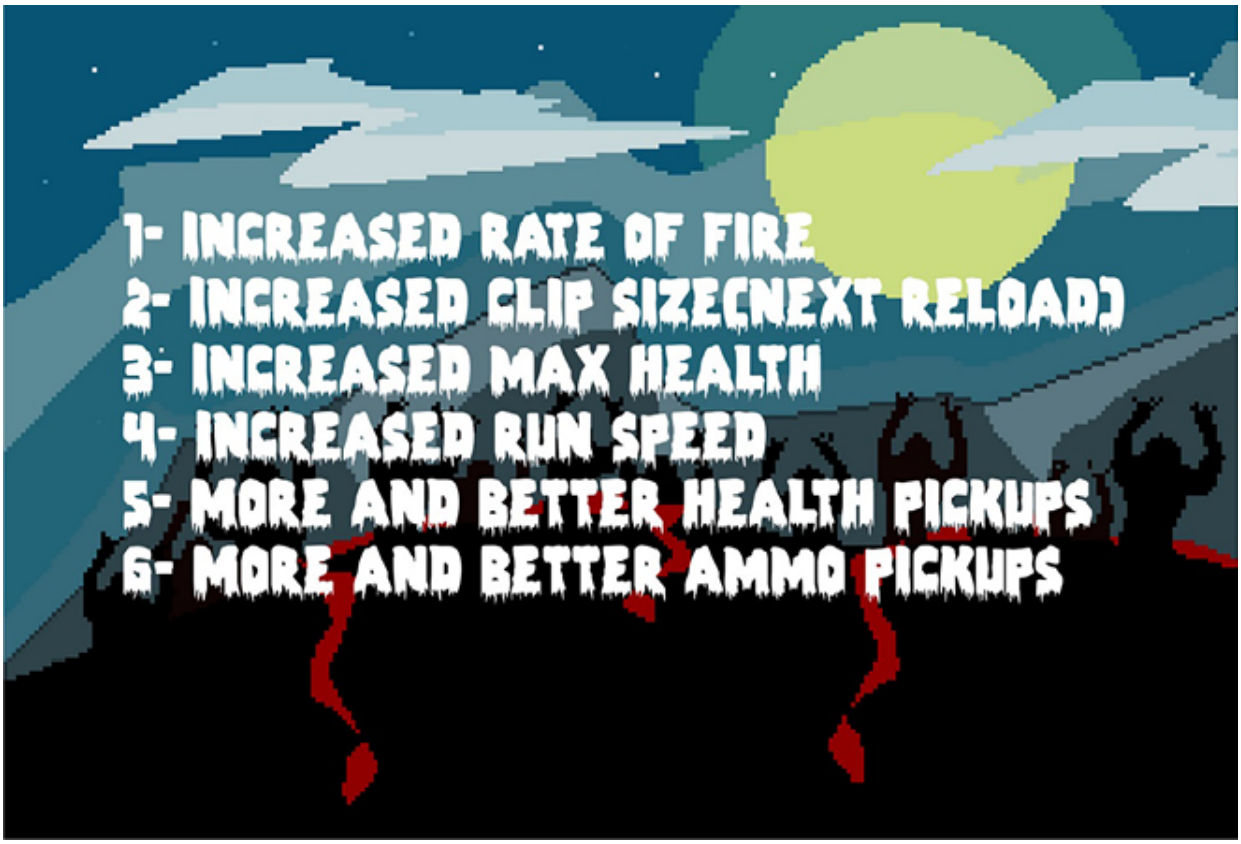

1- INCREASED RATE OF FIRE
2- INCREASED CLIP SIZE(NEXT RELOAD)
3- INCREASED MAX HEALTH
4- INCREASED RUN SPEED
5- MORE AND BETTER HEALTH PICKUPS
6- MORE AND BETTER AMMO PICKUPS

Figure 13.3: Text telling the player what their level-up options are

Here, we can see a helpful message on the pause screen prompting the player to start a new game:

Figure 13.4: Message on the pause screen prompting the player to start a new game

SFML Views are more powerful than this simple HUD can demonstrate. For an insight into the potential of the SFML `View` class and how easy they are to use, look at the SFML website's tutorial on `View` at https://www.sfml-dev.org/tutorials/2.5/graphics-view.php. Furthermore, in the final project, we will use multiple `View` instances to create a mini-map feature.

It is hopefully satisfying to see our game taking shape. The menus are like the glue that holds all the other parts together and make the game playable.

But we still have more to do, so let's keep going.

Summary

This was a quick and simple chapter. We looked at how to display the values that are held by variables of different types using `sstream` and then learned how to draw them over the top of the main game action using a second SFML `View` object.

We are nearly done with Zombie Arena now. We have added and seen how to update the HUD including a level-up and a home screen. All the screenshots in this chapter show a small arena that doesn't take advantage of the full monitor.

In the next chapter, the final one for this project, we will put in some finishing touches, such as leveling up, sound effects, and saving the high score. The arena can then grow to the same size as the monitor and far beyond.

14

Sound Effects, File I/O, and Finishing the Game

We are nearly done with this project. This short chapter will demonstrate how we can easily manipulate files stored on the hard drive using the **C++ Standard Library**, and we will also add sound effects. Of course, we know how to add sound effects, but we will discuss exactly where the calls to the `play` function will go in the code. We will also tie up a few loose ends to make the game complete.

In this chapter, we will cover the following topics:

- Saving and loading the high score
- Preparing sound effects
- Allowing the player to level up and spawning a new wave
- Restarting the game
- Playing the rest of the sounds

Saving and loading the high score

File **I/O** or **input/output** is a fairly technical subject. Fortunately for us, as it is such a common requirement in programming, there is a library that handles all this complexity for us. Like concatenating strings for our HUD,

it is the **C++ Standard Library** that provides the necessary functionality through `fstream`.

First, we include `fstream` in the same way we included `sstream`:

```
#include <sstream>
#include <fstream>
#include <SFML/Graphics.hpp>
#include "ZombieArena.h"
#include "Player.h"
#include "TextureHolder.h"
#include "Bullet.h"
#include "Pickup.h"
using namespace sf;
```

Now, add a new folder to the `ZombieArena` folder called `gamedata`. Next, right-click in this folder and create a new file called `scores.txt`. It is in this file that we will save the player's high score. You can easily open the file and add a score to it. If you do, make sure it is quite a low score so that we can easily test whether beating that score results in the new score being added. Be sure to close the file once you are done with it or the game will not be able to access it.

In the following code, we will create an `ifstream` object called `inputFile` and pass the folder and file we just created as a parameter to its constructor.

`if(inputFile.is_open())` checks that the file exists and is ready to read from. We then put the contents of the file into `hiScore` and close the file. Add the following highlighted code:

```
// Score
Text scoreText;
scoreText.setFont(font);
scoreText.setCharacterSize(55);
```

```
scoreText.setColor(Color::White);
scoreText.setPosition(20, 0);
// Load the high score from a text file
std::ifstream inputFile("gamedata/scores.txt");
if (inputFile.is_open())
{
    // >> Reads the data
    inputFile >> hiScore;
    inputFile.close();
}
// Hi Score
Text hiScoreText;
hiScoreText.setFont(font);
hiScoreText.setCharacterSize(55);
hiScoreText.setColor(Color::White);
hiScoreText.setPosition(1400, 0);
std::stringstream s;
s << "Hi Score:" << hiScore;
hiScoreText.setString(s.str());
```

Now, we can handle saving a potentially new high score. Within the block that handles the player's health being less than or equal to zero, we need to create an `ofstream` object called `outputFile`, write the value of `hiScore` to the text file, and then close the file, like so:

```
// Have any zombies touched the player
for (int i = 0; i < numZombies; i++)
{
    if (player.getPosition().intersects
        (zombies[i].getPosition()) && zombies[i].isAlive())
    {
        if (player.hit(gameTimeTotal))
        {
            // More here later
        }
        if (player.getHealth() <= 0)
        {
            state = State::GAME_OVER;
```

```
            std::ofstream outputFile("gamedata/scores.txt");
            // << writes the data
            outputFile << hiScore;
            outputFile.close();

        }
    }
}// End player touched
```

You can play the game and your high score will be saved. Quit the game and notice that your high score is still there if you play it again.

In the next section, we will make some noise.

Preparing sound effects

In this section, we will create all the `SoundBuffer` and `Sound` objects that we need to add a range of sound effects to the game.

Start by adding the required SFML `#include` statements:

```
#include <sstream>
#include <fstream>
#include <SFML/Graphics.hpp>
#include <SFML/Audio.hpp>
#include "ZombieArena.h"
#include "Player.h"
#include "TextureHolder.h"
#include "Bullet.h"
#include "Pickup.h"
```

Now, go ahead and add the seven `SoundBuffer` and `Sound` objects that load and prepare the seven sound files that we prepared in *Chapter 8*, *SFML Views – Starting the Zombie Shooter Game*:

```
// When did we last update the HUD?
int framesSinceLastHUDUpdate = 0;
// What time was the last update
Time timeSinceLastUpdate;
// How often (in frames) should we update the HUD
int fpsMeasurementFrameInterval = 1000;
// Prepare the hit sound
SoundBuffer hitBuffer;
hitBuffer.loadFromFile("sound/hit.wav");
Sound hit;
hit.setBuffer(hitBuffer);
// Prepare the splat sound
SoundBuffer splatBuffer;
splatBuffer.loadFromFile("sound/splat.wav");
Sound splat;
splat.setBuffer(splatBuffer);
// Prepare the shoot sound
SoundBuffer shootBuffer;
shootBuffer.loadFromFile("sound/shoot.wav");
Sound shoot;
shoot.setBuffer(shootBuffer);
// Prepare the reload sound
SoundBuffer reloadBuffer;
reloadBuffer.loadFromFile("sound/reload.wav");
Sound reload;
reload.setBuffer(reloadBuffer);
// Prepare the failed sound
SoundBuffer reloadFailedBuffer;
reloadFailedBuffer.loadFromFile("sound/reload_failed.wav");
Sound reloadFailed;
reloadFailed.setBuffer(reloadFailedBuffer);
// Prepare the powerup sound
SoundBuffer powerupBuffer;
powerupBuffer.loadFromFile("sound/powerup.wav");
Sound powerup;
powerup.setBuffer(powerupBuffer);
// Prepare the pickup sound
SoundBuffer pickupBuffer;
pickupBuffer.loadFromFile("sound/pickup.wav");
Sound pickup;
pickup.setBuffer(pickupBuffer);
```

```
// The main game loop
while (window.isOpen())
```

Now, the seven sound effects are ready to play. We just need to work out where in our code each of the calls to the `play` function will go.

Allowing the player to level up and spawning a new wave

In the following code, we allow the player to level up between waves. Because of the work we have already done, this is straightforward to achieve.

Add the following highlighted code to the `LEVELING_UP` state where we handle player input:

```
// Handle the LEVELING up state
if (state == State::LEVELING_UP)
{
    // Handle the player LEVELING up
    if (event.key.code == Keyboard::Num1)
    {
        // Increase fire rate
        fireRate++;
        state = State::PLAYING;
    }
    if (event.key.code == Keyboard::Num2)
    {
        // Increase clip size
        clipSize += clipSize;
        state = State::PLAYING;
    }
    if (event.key.code == Keyboard::Num3)
    {
        // Increase health
```

```
            player.upgradeHealth();
            state = State::PLAYING;
        }
        if (event.key.code == Keyboard::Num4)
        {
            // Increase speed
            player.upgradeSpeed();
            state = State::PLAYING;
        }
        if (event.key.code == Keyboard::Num5)
        {
            // Upgrade pickup
            healthPickup.upgrade();
            state = State::PLAYING;
        }
        if (event.key.code == Keyboard::Num6)
        {
            // Upgrade pickup
            ammoPickup.upgrade();
            state = State::PLAYING;
        }
        if (state == State::PLAYING)
        {
```

The player can now level up each time a wave of zombies is cleared. We can't, however, increase the number of zombies or the size of the level just yet.

In the next part of the `LEVELING_UP` state, right after the code we have just added, amend the code that runs when the state changes from `LEVELING_UP` to `PLAYING`.

The following is the code in full. I have highlighted the lines that are either new or have been slightly amended. Add or amend the following highlighted code:

```
    if (event.key.code == Keyboard::Num6)
    {
        ammoPickup.upgrade();
        state = State::PLAYING;
    }
    if (state == State::PLAYING)
    {
        // Increase the wave number
        wave++;
        // Prepare the level
        // We will modify the next two lines later
        arena.width = 500 * wave;
        arena.height = 500 * wave;
        arena.left = 0;
        arena.top = 0;
        // Pass the vertex array by reference
        // to the createBackground function
        int tileSize = createBackground(background, arena);
        // Spawn the player in the middle of the arena
        player.spawn(arena, resolution, tileSize);
        // Configure the pick-ups
        healthPickup.setArena(arena);
        ammoPickup.setArena(arena);
        // Create a horde of zombies
        numZombies = 5 * wave;
        // Delete the previously allocated memory (if it exists
        delete[] zombies;
        zombies = createHorde(numZombies, arena);
        numZombiesAlive = numZombies;
        // Play the powerup sound
        powerup.play();
        // Reset the clock so there isn't a frame jump
        clock.restart();
    }
}// End LEVELING up
```

The previous code starts by incrementing the `wave` variable. Then the code is amended to make the number of zombies and size of the arena relative to the new value of `wave`. This is also useful because the game probably was a

bit hard with 10 zombies in a small area. Now it will start with 5. Finally, we add the call to `powerup.play()` to play the "leveling up" sound effect.

Restarting the game

We have already determined the size of the arena and the number of zombies by the value of the `wave` variable. We must also reset the ammo and gun-related variables, and set `wave` and `score` to zero at the start of each new game.

Find the following code in the event-handling section of the game loop and add the highlighted code, as shown here:

```
// Start a new game while in GAME_OVER state
else if (event.key.code == Keyboard::Return &&
    state == State::GAME_OVER)
{
    state = State::LEVELING_UP;
    wave = 0;
    score = 0;
    // Prepare the gun and ammo for next game
    currentBullet = 0;
    bulletsSpare = 24;
    bulletsInClip = 6;
    clipSize = 6;
    fireRate = 1;
    // Reset the player's stats
    player.resetPlayerStats();
}
```

Now, players can engage in the game, becoming increasingly powerful as the number of zombies grows within an ever-expanding arena. The game continues until the player dies, after which it starts over again.

Playing the rest of the sounds

Now, we will add the rest of the calls to the `play` function. We will address each of them individually, as pinpointing exactly where they go in the code is crucial to using them at the right moment.

Adding sound effects while the player is reloading

Add the following highlighted code in three specific locations to trigger the appropriate reload or `reloadFailed` sound when the player presses the *R* key to attempt reloading their gun:

```
Tif (state == State::PLAYING)
{
    // Reloading
    if (event.key.code == Keyboard::R)
    {
        if (bulletsSpare >= clipSize)
        {
            // Plenty of bullets. Reload.
            bulletsInClip = clipSize;
            bulletsSpare -= clipSize;
            reload.play();
        }
        else if (bulletsSpare > 0)
        {
            // Only few bullets left
            bulletsInClip = bulletsSpare;
            bulletsSpare = 0;
            reload.play();
        }
        else
        {
            // More here soon?!
            reloadFailed.play();
```

```
        }
    }
}
```

The player will now get an audible response when they reload or attempt to reload the gun. Let's move on to playing a shooting sound.

Making a shooting sound

Add the following highlighted call to `shoot.play()` near the end of the code that handles the player clicking the left mouse button:

```
// Fire a bullet
if (sf::Mouse::isButtonPressed(sf::Mouse::Left))
{
    if (gameTimeTotal.asMilliseconds()
        - lastPressed.asMilliseconds()
        > 1000 / fireRate && bulletsInClip > 0)
    {
        // Pass the centre of the player and crosshair
        // to the shoot function
        bullets[currentBullet].shoot(
            player.getCenter().x, player.getCenter().y,
            mouseWorldPosition.x, mouseWorldPosition.y);
        currentBullet++;
        if (currentBullet > 99)
        {
            currentBullet = 0;
        }
        lastPressed = gameTimeTotal;
        shoot.play();
        bulletsInClip--;
    }
}// End fire a bullet
```

The game will now play a satisfying shooting sound. Next, we will play a sound when the player is hit by a zombie.

Playing a sound when the player is hit

In this following code, we wrap the call to `hit.play` in a test to see if the `player.hit` function returns true. Remember that the `player.hit` function tests to see if a hit has been recorded in the previous 100 milliseconds. This will have the effect of playing a fast-repeating thud sound, but not so fast that the sound blurs into a single noise.

Add the call to `hit.play`, as highlighted in the following code:

```
// Have any zombies touched the player
for (int i = 0; i < numZombies; i++)
{
    if (player.getPosition().intersects
        (zombies[i].getPosition()) && zombies[i].isAlive())
    {
        if (player.hit(gameTimeTotal))
        {
            // More here later
            hit.play();
        }
        if (player.getHealth() <= 0)
        {
            state = State::GAME_OVER;
            std::ofstream OutputFile("gamedata/scores.txt");
            OutputFile << hiScore;
            OutputFile.close();

        }
    }
}// End player touched
```

The player will hear an ominous thudding sound when a zombie touches them, and this sound will repeat around five times per second if the zombie continues touching them. The logic for this is contained in the `hit` function of the `Player` class.

Playing a sound when getting a pickup

When the player picks up a health pickup, we will play the regular pickup sound. However, when the player gets an ammo pickup, we will play the reload sound effect.

Add the two calls to play sounds within the appropriate collision detection code:

```
// Has the player touched health pickup
if (player.getPosition().intersects
    (healthPickup.getPosition()) && healthPickup.isSpawned())
{
    player.increaseHealthLevel(healthPickup.gotIt());
    // Play a sound
    pickup.play();

}
// Has the player touched ammo pickup
if (player.getPosition().intersects
    (ammoPickup.getPosition()) && ammoPickup.isSpawned())
{
    bulletsSpare += ammoPickup.gotIt();
    // Play a sound
    reload.play();

}
```

Making a splat sound when a zombie is shot

Add a call to `splat.play` at the end of the section of code that detects a bullet colliding with a zombie:

```
// Have any zombies been shot?
for (int i = 0; i < 100; i++)
{
    for (int j = 0; j < numZombies; j++)
    {
        if (bullets[i].isInFlight() &&
            zombies[j].isAlive())
        {
            if (bullets[i].getPosition().intersects
                (zombies[j].getPosition()))
            {
                // Stop the bullet
                bullets[i].stop();
                // Register the hit and see if it was a kill
                if (zombies[j].hit()) {
                    // Not just a hit but a kill too
                    score += 10;
                    if (score >= hiScore)
                    {
                        hiScore = score;
                    }
                    numZombiesAlive--;
                    // When all the zombies are dead (again)
                    if (numZombiesAlive == 0) {
                        state = State::LEVELING_UP;
                    }
                }
                // Make a splat sound
                splat.play();

            }
        }
```

```
        }
    }// End zombie being shot
```

You can now play the completed game and watch the number of zombies and the arena increase with each wave. Remember to choose your level-ups carefully.

Congratulations!

Summary

We've finished the Zombie Arena game. It has been quite a journey. We have learned a whole bunch of C++ fundamentals, such as references, pointers, OOP, and classes. In addition, we have used SFML to manage cameras (views), vertex arrays, and collision detection. We learned how to use sprite sheets to reduce the number of calls to `window.draw` and speed up the frame rate. By using C++ pointers, the STL, and a little bit of OOP, we built a singleton class to manage our textures.

Frequently asked questions

Here are some problems that might be on your mind:

Q1) Despite using classes, I am finding that the code is getting very long and unmanageable again.

A) One of the biggest issues is the structure of our code. As we learn more C++, we will also learn ways to make the code more manageable and generally less lengthy. We will do so in the next and final project too. By the end of this book, you will know about a number of strategies that you can use to manage your code.

Q2). The sound effects seem a bit flat and unrealistic. How can they be improved?

A) One way to significantly improve the feeling the player gets from sound is to make the sound directional. You can also change the volume based on the distance from the sound source to the player character. We will use SFML's advanced sound features in the next project. Another common trick is to vary the pitch of the gunshot each time as this makes the sound more realistic and less monotonous.

15

Run!

Welcome to the final project: Run! **Run** is an endless runner where the objective of the player is to stay ahead of the disappearing platforms that are catching them up from behind. In this project, we will learn loads of new game programming techniques and even more C++ topics to implement those techniques. Perhaps the best improvement this game will have over the previous games is that it will be way more object oriented than any of the others. There will be many more classes than any of the preceding projects but most of the code files for these classes will be short and uncomplicated. Furthermore, we will build a game where the functionality and appearance of all the in-game objects are pushed out to classes, leaving the main game loop unchanged regardless of what the game objects do. This is powerful because it means you can make a hugely varied game just by designing new stand-alone components (classes) that describe the behavior and appearance of the required game entity. This means you can use the same code structure for a completely different game of your own design. But there is way more to come than just this. Read on for the details.

The completed code for this chapter can be found in the `Run` folder.

Here is what we will cover in this chapter:

- Describing exactly what the game is and how it will be played.

- Creating the project in the usual way and coding the simplest main function of the entire book!
- Discussing and coding the new way we will handle the player's input by delegating specific responsibilities to individual game entities/objects and having them listen for messages from a new `InputDispatcher` class.
- Coding a class called `Factory`, which will be responsible for "knowing" how to assemble all the different components we will build into usable `GameObject` instances.
- Learning about C++ inheritance and polymorphism is not as hard as it sounds.
- Learning about C++ smart pointers for passing responsibility for memory management to the compiler.
- Coding the key `GameObject` class; you won't believe how short and simple this is.
- Coding the `Component` class, which the `GameObject` instances will hold. Again, this is short and simple.
- Coding the Graphics and Update classes, which will be types of `Component`. This will make more sense when we learn about inheritance and polymorphism.
- Finally, to end the chapter, we will have a functioning game loop that listens for player input and draws a blank screen so it is ready for all the parts we will code for the remainder of the book.

First, we need to know what we are going to build. At the same time, I will introduce all the new game programming concepts we will learn.

You will find this chapter's source code in the GitHub repository: https://github.com/PacktPublishing/Beginning-C-

Game-Programming-Third-Edition/tree/main/Run

About the game

Run! is a very simple game. In fact, it is the fewest number of game entities I could think of that could be considered a playable game. I designed the game around demonstrating a reusable system for game development rather than compelling gameplay. This makes the project ideal for you to add new behavior, rules, and gameplay to your own design. Or, even better, once you have learned how it works, design a completely new game of your own using the system as well as improve and add features to the system.

The system we will build is a version of the **entity component** programming pattern. A pattern is a way to do things. We will discuss the entity component pattern some more once we have discussed **inheritance** and **polymorphism**. For now, let's see the game. The following screenshots show most of the entities that make up our game:

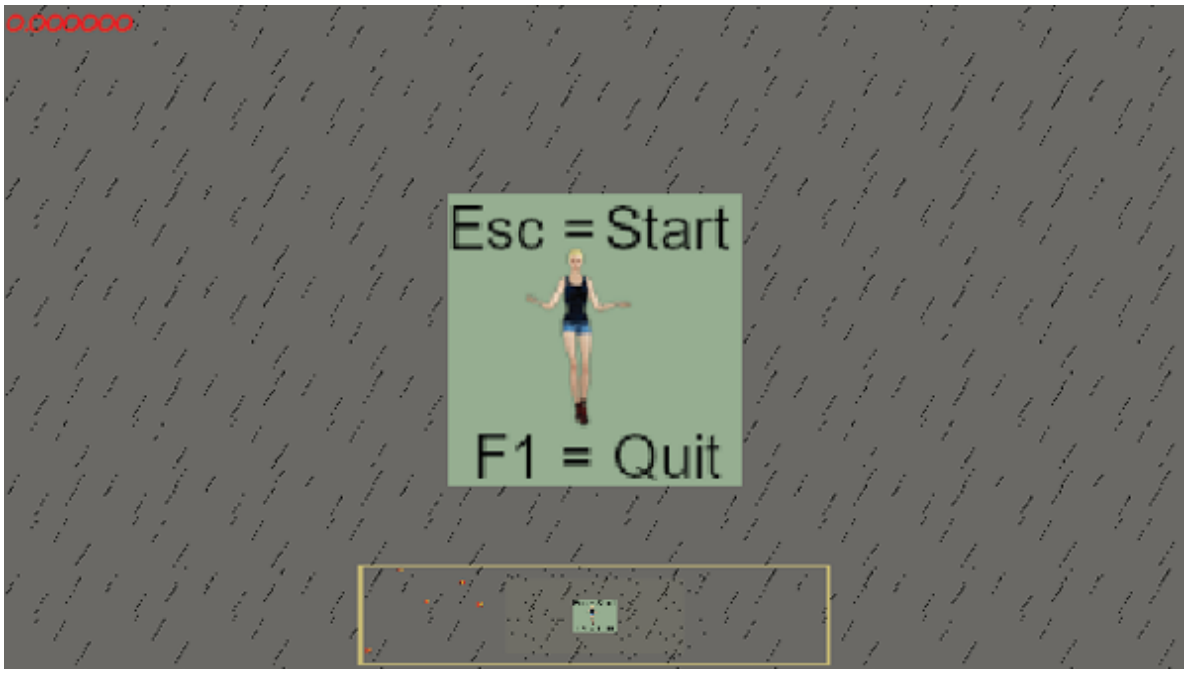


Figure 15.1: The game menu

In the preceding image, you can see a simple game menu. The player can press the *Esc* key to start or pause the game and the *F1* key to quit. When the game is started, a timer in the top left of the screen is started and the objective of the game is to last as long as possible by running to the right and keeping up with new platforms that are constantly spawning on the right and keeping away from the constantly disappearing platforms on the left. When the disappearing platforms catch up with the player, the game ends and the menu will be shown again. Look at the next image:

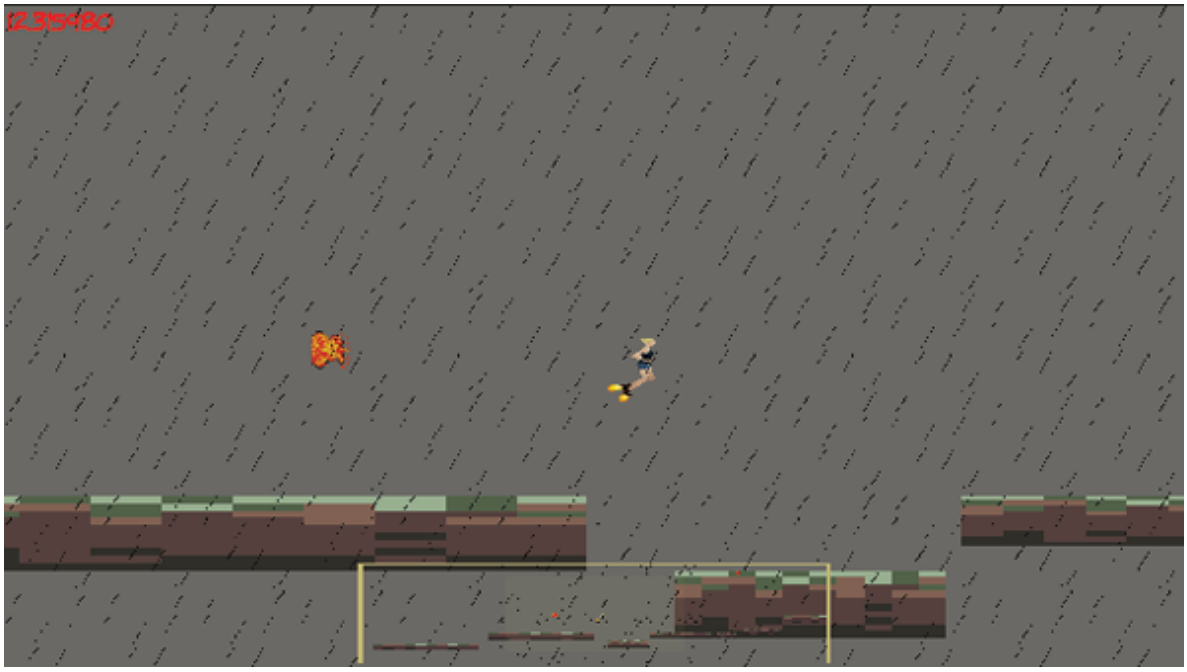


Figure 15.2: Game rain

In the preceding image, you can see the player character in the center of the screen. Fire appears to be coming out of her boots. This is the effect that shows that the player is boosting. If the player falls off the platforms, they can press the *W* key to boost upwards. Additionally, the player can travel left and right while boosting by pressing the *A* and *D* keys respectively. Be

aware that while boosting, horizontal movement is slow, and the disappearing platforms will catch up much faster than while running and jumping. Running is also achieved with the *A* and *D* keys. Note that you will almost always be running to the right with the *D* key. The spacebar is the key used to jump between platforms. Boosting is a short-term emergency measure to correct a missed jump, not a strategy for beating the game. Running and jumping are fast and lead to survival: boosting is slow and leads to death.

In the preceding screenshot, you can also see that the fireball is directly to the left of the player. Fireballs knock the player downward, usually causing the player to fall below the platform they are on and forcing them to boost to survive. Fireballs are spawned randomly throughout the game and can come from the left or right. As fireballs are fast, the player will get two advanced warnings of an incoming fireball. First, as a fireball is fired at the player, a roaring sound will be played from either the right or the left using spatialized/directional sound effects. Secondly, notice the minimap in the bottom center of the screen, which shows an area of the world much wider than the main screen. The player will be able to glance at this radar-like minimap and see whether the incoming fireball is on a collision path and take early evasive action if required.

Also, in the preceding image, you can see that there is a simple rain effect. Look at this next image to see some more features of the game. This might be unclear if you are looking at a black-and-white image in the paperback edition:

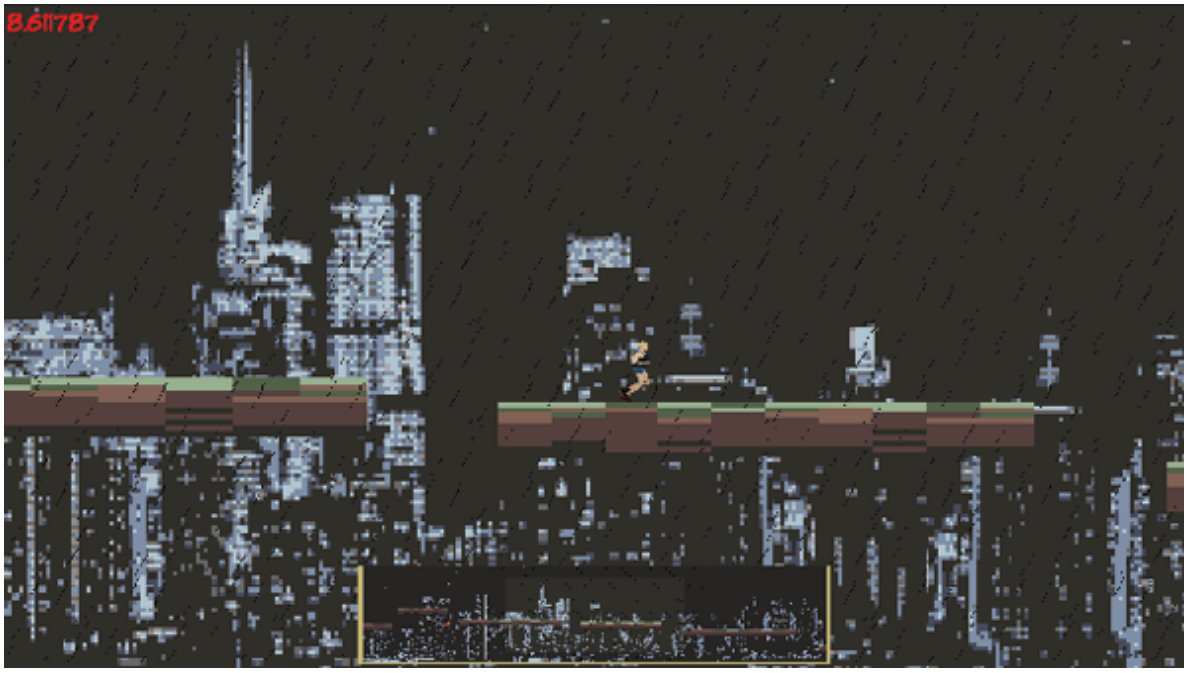


Figure 15.3: Game parallax

In the preceding image, there is a kind of night-time cityscape background. The background will scroll left and right with the player's movement, but it will move more slowly than the platforms in the foreground. This will create a parallax effect, giving the impression the city is in the distance.

Figure 15.4: Game shader

In the preceding image, we see a complete change in the appearance of the background. By using **OpenGL** shaders, we can achieve an almost photographic 3D rolling countryside effect. Amazingly, we will add this effect with only a few lines of C++ code, but the shader program itself is quite a complicated program we will obtain from a website that specializes in cool shaders. We will explore how shaders work, how to use them, and what they are, but we won't explore how to code them for ourselves.

Creating the project

We need to create a new project. To get started, create a new project, put it in the `VS Projects` folder, call it `Run`, and copy the `fonts`, `graphics`, `music`, `shaders`, `sound` folders and their contents into the project folder. We will discuss the folder contents as we proceed. There are some significant differences in the assets compared to previous projects: the shaders, for one,

the music, for another, and the fact that there is just a single image file in the `graphics` folder that contains all the visuals for the entire game. The files in the `shaders` folder are empty placeholder files ready to have some publicly available code copied and pasted into them later in the project.

I have created a working project for each chapter so you can refer to what the code should look like at the end of each chapter. You will see the project folders called `Run`, `Run2`, `Run3`, and so on. You can therefore see the completed code for this chapter in the `Run` project folder.

You do not need to go to the trouble of creating a new project for each chapter! The instructions for each chapter flow neatly from the preceding chapter.

Configure the project properties as we have done for all the previous projects. What follows is an abbreviated reminder of how to do this. For images and more details, refer to *Chapter 1*. Now, complete the following steps:

1. We will now configure the project to use the SFML files that we put in the `SFML` folder. From the main menu, select **Project | Properties**…. At this stage, you should have the **Run Property Pages** window open.
2. In the **Run Property Pages** window, take the following steps. Select **All Configurations** from the **Configuration:** drop-down menu and make sure the drop-down menu to the right is set to **Win32**, not **x64**.
3. Now, select **C/C++** and then **General** from the left-hand menu.
4. Next, locate the **Additional Include Directories** edit box and type the drive letter where your `SFML` folder is located, followed by `\SFML\include`. The full path to type, if you located your `SFML` folder on your D drive, will be `D:\SFML\include`. Vary your path if you installed SFML on a different drive.

5. Now, still in the same window, perform these next steps. From the left-hand menu, select **Linker** and then **General**.
6. Now, find the **Additional Library Directories** edit box and type the drive letter where your `SFML` folder is, followed by `\SFML\lib`. So, the full path to type, if you located your `SFML` folder on your D drive, will be `D:\SFML\lib`. Change your path if you installed SFML on a different drive.
7. Select **Linker** and then **Input**.
8. Find the **Additional Dependencies** edit box and click on it on the far left-hand side. Now, copy and paste or type the following: `sfml-graphics-d.lib;sfml-window-d.lib;sfml-system-d.lib;sfml-network-d.lib;sfml-audio-d.lib;`. Be extra careful to place the cursor exactly at the start of the edit box's current content so as not to overwrite any of the text that is already there.
9. Click on **OK**.
10. Click on **Apply** and then **OK**.
11. Back in the main Visual Studio screen, check that the main menu toolbar is set to **Debug** and **x86**, not **x64**.
12. Finally, copy all of the `sfml-…-d-2.dll` files into the project directory where `…` refers to `audio`, `graphics`, `network`, `system`, and `window`.

Now, we will move on to the C++ code.

Coding the main function

What follows is all the code for the `main` function. It is also the entire game loop. There is no collision detection, no pause, start, or stop logic, no sprites or textures, no fonts or sounds, and only one line relates to handling input. In this project, everything, or almost everything, will be a game object.

Cameras, fireballs, platforms, the player character, the menu, and even the game logic and the rain will be game objects. Exactly how this is achieved is explained over the course of the project, but a good overview will be given in the *Entity Component System pattern* section later in this chapter. For now, let's make some progress with the code.

Add the following code to the `run.cpp` file in your project and we will then go through it one section at a time:

```
#pragma once
#include "SFML/Graphics.hpp"
#include <vector>
#include "GameObject.h"
#include "Factory.h"
#include "InputDispatcher.h"
using namespace std;
using namespace sf;
int main()
{
  // Create a fullscreen window.
RenderWindow window(
    VideoMode::getDesktopMode(),
 "Booster", Style::Fullscreen);
  // A VertexArray to hold all our graphics.
VertexArray canvas(Quads, 0);
  // This can dispatch events to any object.
InputDispatcher inputDispatcher(&window);
  // Everything will be a game object.
// This vector will hold them all.
  vector <GameObject> gameObjects;
  // This class has all the knowledge
// to construct game objects that do
// many different things.
Factory factory(&window);
  // This call will send the vector of game objects
// the canvas to draw on and the input dispatcher
// to the factory to set up the game.
  factory.loadLevel(gameObjects,
    canvas,
```

```
    inputDispatcher);
  // A clock for timing.
  Clock clock;
  // The color we use for the background
const Color BACKGROUND_COLOR(100, 100, 100, 255);
  // This is the game loop.
// We do not need to add to it.
// Look how short and simple it is!
while (window.isOpen())
  {
    // Measure the time taken this frame.
float timeTakenInSeconds =
      clock.restart().asSeconds();
    // Handle the player input.
    inputDispatcher.dispatchInputEvents();
    // Clear the previous frame.
    window.clear(BACKGROUND_COLOR);
    // Update all the game objects.
for (auto& gameObject : gameObjects)
    {
      gameObject.update(timeTakenInSeconds);
    }
    // Draw all the game objects to the canvas.
for (auto& gameObject : gameObjects)
    {
      gameObject.draw(canvas);
    }
    // Show the new frame.
    window.display();
  }
  return 0;
}
```

First, note that there are three errors showing in Visual Studio. This is because we are referencing three classes that don't exist yet. The missing classes are `InputDispatcher`, `GameObject`, and `Factory`. We will code these classes soon, but first, let's discuss the code, or, compared to previous projects, the lack of code. The code starts with this:

```
#pragma once
#include "SFML/Graphics.hpp"
#include <vector>
#include "GameObject.h"
#include "Factory.h"
#include "InputDispatcher.h"
using namespace std;
using namespace sf;
```

The preceding code has the usual SFML include directive and another one for the `vector` class. This implies we will have a vector in our code. We will use a vector to hold all our game objects. Furthermore, we have three includes for `GameObject`, `Factory`, and `InputDispatcher`, which will have errors until we code them later in the chapter.

Observe the first part of the `main` function:

```
int main()
{
  // Create a fullscreen window.
RenderWindow window(
    VideoMode::getDesktopMode(),
 "Booster", Style::Fullscreen);
  // A VertexArray to hold all our graphics.
VertexArray canvas(Quads, 0);
  // This can dispatch events to any object.
InputDispatcher inputDispatcher(&window);
  // Everything will be a game object.
// This vector will hold them all.
  vector <GameObject> gameObjects;
  // This class has all the knowledge
// to construct game objects that do
// many different things.
Factory factory(&window);
// This call will send the vector of game objects
// the canvas to draw on and the input dispatcher
// to the factory to set up the game.
  factory.loadLevel(gameObjects,
```

```
        canvas,
        inputDispatcher);
    // A clock for timing.
    Clock clock;
    // The color we use for the background
const Color BACKGROUND_COLOR(100, 100, 100, 255);
```

In the preceding code, we created a `RenderWindow` instance as we have for all our games. We create an SFML `VertexArray` instance called `canvas`. We call the `VertexArray` `canvas` because it will literally be the canvas for the entire game. All the game objects will be added to the `VertexArray` each frame of the game and then `canvas` will be used to draw to the window. Next up, we declare an instance of our forthcoming `InputDispatcher` class. We will see how that gets used in the main game loop soon. For now, just notice that we send the address of `RenderWindow` to its constructor. Next, we declare a `vector` of the `GameObject` instances. As already stated, every entity in our game will be contained in a `GameObject` instance. Exactly how this is possible will be revealed as we proceed. The next two lines of code declare an instance of our soon-to-be-coded `Factory` class (which also gets a pointer to the `RenderWindow`) and then we call `factory.loadLevel`. The `loadLevel` function requires the `vector` of game objects, the canvas to draw on, and the `InputDispatcher` instance. The `Factory` class will be the part of our game engine that assembles the wide array of `GameObject` instances in the correct manner and the correct order and then places them in the vector ready for use in the game loop.

Lastly in the code we are currently discussing, we declare a clock to handle the timing of updates and a color to draw as a temporary background.

Look at the `main` loop in the following code again and we will then go through it:

```
while (window.isOpen())
{
  // Measure the time taken this frame.
float timeTakenInSeconds =
  clock.restart().asSeconds();
  // Handle the player input.
  inputDispatcher.dispatchInputEvents();
// Clear the previous frame.
  window.clear(BACKGROUND_COLOR);
  // Update all the game objects.
for (auto& gameObject : gameObjects)
  {
    gameObject.update(timeTakenInSeconds);
  }
  // Draw all the game objects to the canvas.
for (auto& gameObject : gameObjects)
  {
    gameObject.draw(canvas);
  }
  // Show the new frame.
  window.display();
}
```

In the preceding code, we have the usual `while` loop to constantly loop through updating and drawing the game objects until the window is closed. The duration of the loop is captured in the `timeTakenInSeconds` variable and then we see something new.

The `inputDispatcher` instance calls the `dispatchInputEvents` function. Inside this function, which we will code shortly, all of the input events are shared with any game object that has previously declared an interest. The `Factory` class takes care of allowing game objects to connect with the `inputDispatcher` and then each game object handles the inputs that it cares about. So, the player character will handle movement, the menu will handle pausing, starting, and quitting, and we will even have a camera game object that handles scrolling the mouse wheel to zoom in and out of the minimap.

Next, we have two `for` loops that loop through all the game objects in the vector, first calling `update` and then calling `draw`. Once the canvas is updated, `window.display()` shows the entire game in its current state.

The `main` function then ends as follows:

```
    return 0;
}
```

Now that we have seen what we are aiming for, we will write two new classes to make the new, more flexible input system work.

Handling input

In the preceding code, you will notice a distinct lack of input handling code. This is because each game object will be responsible for handling its own **input events**. Most notable is the player-related game object that will handle movement input from the player.

There will also be a menu-related game object that will handle starting, pausing, and quitting the game and a camera-related object that will represent the minimap/radar that the player will be able to zoom in and out of. The point is that each object will handle its own input events. This next image illustrates this setup:

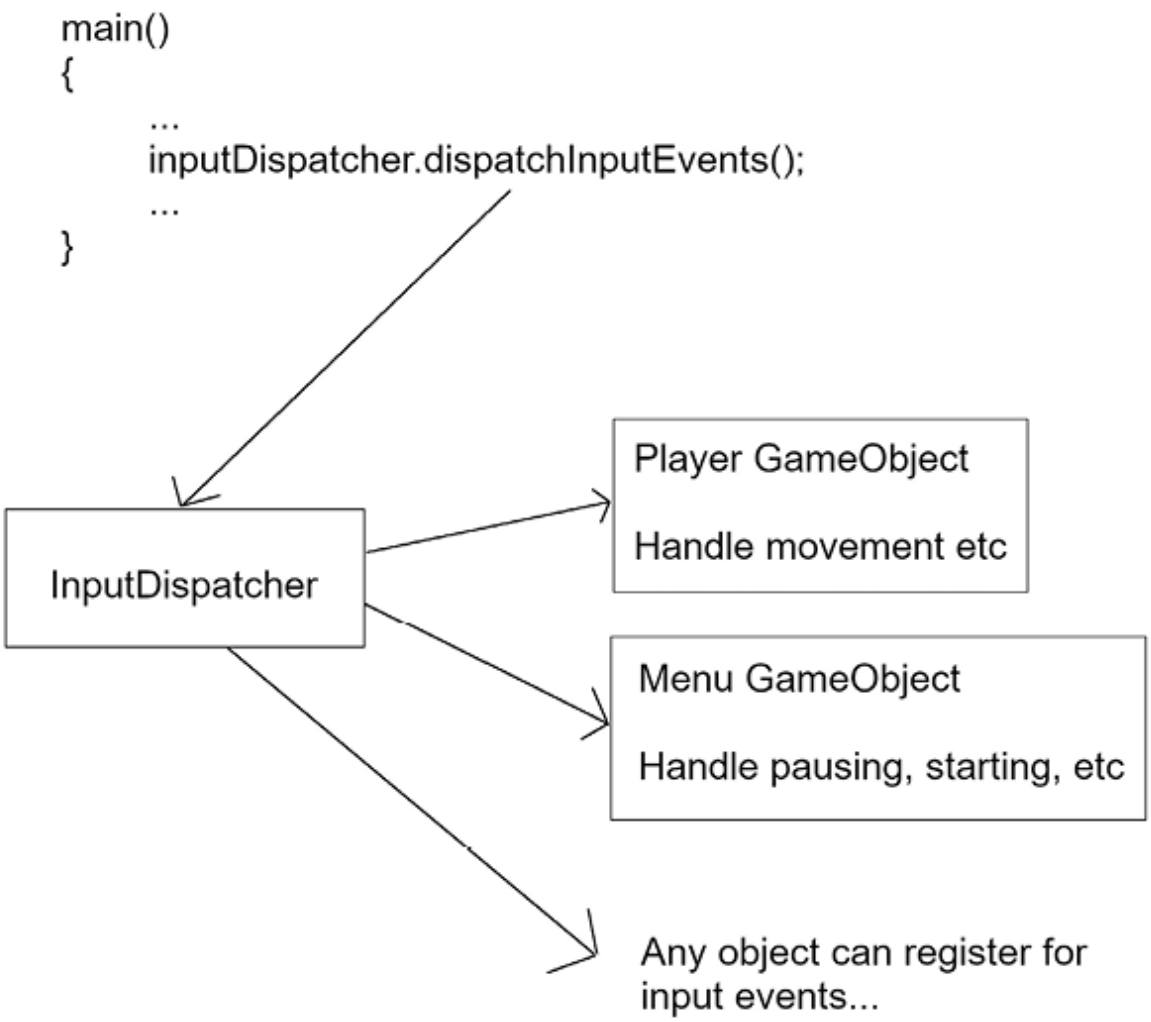


Figure 15.5: Handling input diagram

To achieve this, we will code an `InputDispatcher` class of which, as we have seen in the `main` function, there will be a single instance that will receive all the input events from the operating system and then dispatch them to several `InputReceiver` class instances, which will have previously made themselves known (registered) to the `InputDispatcher` instance during the `loadLevel` function execution in the `Factory` class, before the main game loop. All the `InputReceiver` instances will be inside an appropriate game object that will know what input events to watch out for and how to handle them.

Let's code the `InputDispatcher` class. Create a new class called `InputDispatcher`. First, add the following code to `InputDispatcher.h`:

```
#pragma once
#include "SFML/Graphics.hpp"
#include "InputReceiver.h"
using namespace sf;
class InputDispatcher
{
private:
    RenderWindow* m_Window;
    vector <InputReceiver*> m_InputReceivers;
public:
    InputDispatcher(RenderWindow* window);
    void dispatchInputEvents();
    void registerNewInputReceiver(InputReceiver* ir);
};
```

First, note there is an error because we reference the `InputReceiver` class that we haven't coded yet. No problem; we will get to that as soon as we finish this class.

In the preceding code, we declare a pointer to a `RenderWindow` instance and a `vector` of `InputReceiver` pointers. Each frame this vector will be iterated, and the inputs received by the window will be shared. We have three functions: the constructor that will set the class up, the `dispatchEvents` function that is called from the game loop each frame, and the `registerNewInputReceiver` function that will add `InputReceiver` instances to the vector of `InputReceiver` pointers.

Of course, seeing the implementation of these functions will make things much clearer. Next, add the following code to `InputDispatcher.cpp`:

```
#include "InputDispatcher.h"
InputDispatcher::InputDispatcher(RenderWindow* window)
```

```
{
    m_Window = window;
}
void InputDispatcher::dispatchInputEvents()
{
    sf::Event event;
    while (m_Window->pollEvent(event))
    {
        //if (event.type == Event::KeyPressed &&
//    event.key.code == Keyboard::Escape)
//{
//    m_Window->close();
//}
for (const auto& ir : m_InputReceivers)
        {
            ir->addEvent(event);
        }
    }
}
void InputDispatcher::registerNewInputReceiver(InputReceiver* ir
{
    m_InputReceivers.push_back(ir);
}
```

Again, there are a few errors in the class that are due to the absence of the `InputReceiver` class.

In the preceding code, the constructor initializes the pointer to `RenderWindow`. In the `dispatchInputEvents` function, the `RenderWindow` instance is used to poll all the events in the same way we have done in every project so far. Then, all of the `InputDispatcher` instances in the vector are iterated and their `addEvent` functions are called with the latest event. There is some commented-out code in this function that we will uncomment temporarily later in the chapter. The `registerNewInputReceiver` function allows the code that calls it to pass in a pointer to an `InputReceiver` and therefore receive all the updates. Remember that the `Factory` class receives

the `InputDispatcher` instance when its `loadLevel` function is called. The `loadLevel` function will create all the `InputReceiver` instances, register them with the `register…` function, and place the `InputReceiver` instances inside the appropriate `GameObject` instances.

Let's code `InputReceiver` to see the other side of this system. Create a new class called `InputReceiver`. In the `InputReceiver.h` code file, add the following code:

```
#pragma once
#include <SFML/Graphics.hpp>
using namespace sf;
using namespace std;
class InputReceiver
{
private:
    vector<Event> mEvents;
public:
    void addEvent(Event event);
    vector<Event>& getEvents();
    void clearEvents();
};
```

Notice that all the errors in the `InputDispatcher` class are gone.

The preceding code has a vector of SFML events ready to receive the input events each frame from the input dispatcher. There are three functions: the `addEvent` function receives a new event, the `getEvents` function returns the entire vector full of events, and the `clearEvents` function empties the vector so that events from previous iterations do not build up over time and only the latest events from the current loop are present.

Coding these functions will help us understand all this. Add the following code to the `InputReceiver.cpp` file:

```
#include "InputReceiver.h"
void InputReceiver::addEvent(Event event)
{
    mEvents.push_back(event);
}
vector<Event>& InputReceiver::getEvents()
{
    return mEvents;
}
void InputReceiver::clearEvents()
{
    mEvents.clear();
}
```

In the preceding code, the `addEvent` function uses `pushback` to add an `Event` instance to the vector. The `getEvents` function returns the entire vector to the calling code. Finally, the `clearEvents` function empties the vector so it is ready to receive more events in the next iteration of the game loop. We will see in a later chapter that the appropriate classes will hold an instance of `InputReceiver` and call each of these functions in turn.

Next, let's code the first iteration of the `Factory` class.

Coding the Factory class

Create a new class called `Factory`. In the `Factory.h` file, add the following code.

```
#pragma once
#include <vector>
#include "GameObject.h"
#include "SFML/Graphics.hpp"
using namespace sf;
using namespace std;
class InputDispatcher;
```

```
class Factory
{
private:
  RenderWindow* m_Window;
public:
  Factory(RenderWindow* window);
  void loadLevel(
    vector <GameObject>& gameObjects,
    VertexArray& canvas,
    InputDispatcher& inputDispatcher);
  Texture* m_Texture;
};
```

In the preceding code, the `Factory` class is declared and a private `RenderWindow` pointer is as well.

Note this will be initialized to point to the same `RenderWindow` instance from the `main` function and the same `RenderWindow` instance from the `InputDispatcher` class. There are three functions: the constructor that receives the `RenderWindow` address, the `loadLevel` function that receives the vector of `GameObject` instances, the `VertexArray` for drawing, and the `InputDispatcher` instance pointer. We also declare an instance of an SFML `Texture`. Look at the next two lines of code. They are from the `main` function and are shown here as a reminder of how we call the `Factory` constructor and `loadLevel` functions.

```
Factory factory(&window);
factory.loadLevel(gameObjects,
canvas,
inputDispatcher);
```

After that quick reminder, let's code the `Factory` class functions. In the `Factory.cpp` file, add the following code:

```
#include "Factory.h"
#include <iostream>
using namespace std;
Factory::Factory(RenderWindow* window)
{
  m_Window = window;
  m_Texture = new Texture();
  if (!m_Texture->loadFromFile("graphics/texture.png"))
  {
    cout << "Texture not loaded";
    return;
  }
}
void Factory::loadLevel(
  vector<GameObject>& gameObjects,
  VertexArray& canvas,
  InputDispatcher& inputDispatcher)
{
}
```

In the preceding constructor, we initialize the `RenderWindow` pointer so we always have access to it from this class, specifically from the `loadLevel` function. Furthermore, we load a `.png` file into the texture instance. The file loaded is the one that contains all the graphics for all the game objects. In the next chapter, we will discuss why we do this and how we will make this significant change from previous projects work. The quick explanation is that it is much faster to draw one `VertexArray` compared to the dozens of SFML Sprite instances we have drawn previously.

The `loadLevel` function is left empty for now. We just want to get our code error free by the end of the chapter so we can start achieving significant steps forward with each of the following chapters.

There are still several errors throughout our code, but they are all due to the absence of the `GameObject` class. To fix that, we need to learn some more

C++. Next, we will discuss the modern way to handle pointers as well as some more advanced knowledge about OOP. The next two sections will prepare us to code our `GameObject` class and get the preceding code to run error free.

Advanced OOP: inheritance and polymorphism

In this section, we will further extend our knowledge of OOP by looking at the slightly more advanced concepts of **inheritance** and **polymorphism**. We will then be able to use this new knowledge to implement the game objects and components of our game.

Inheritance

We have already seen how we can use other people's hard work by instantiating objects from the classes of the SFML library. But this whole OOP thing goes even further than that.

What if there is a class that has loads of useful functionality in it, but is not quite what we want? In this situation, we can **inherit** from the other class. Just like it sounds, **inheritance** means we can harness all the features and benefits of other people's classes, including the encapsulation, while further refining or extending the code specifically to our situation. In this project, we will inherit from and extend some of our own classes.

Let's look at some code that uses inheritance.

Extending a class

With all this in mind, let's look at an example class and see how we can extend it, just to see the syntax and as a first step.

First, we define a class to inherit from. This is no different from how we created any of our other classes. Take a look at this hypothetical `Soldier` class declaration:

```
class Soldier
{
    private:
        // How much damage can the soldier take
int m_Health;
        int m_Armour;
        int m_Range;
        int m_ShotPower;


    Public:
        void setHealth(int h);
        void setArmour(int a);
        void setRange(int r);
        void setShotPower(int p);
};
```

In the previous code, we defined a `Soldier` class. It has four private variables: `m_Health`, `m_Armour`, `m_Range`, and `m_ShotPower`. It has also four public functions: `setHealth`, `setArmour`, `setRange`, and `setShotPower`. We don't need to see the definitions of these functions; they will simply initialize the appropriate variable that their name makes obvious.

We can also imagine that a fully implemented `Soldier` class would be much more in depth than this. It would probably have functions such as `shoot` and `goProne`. If we implemented a `Soldier` class in an SFML project, it might have a `Sprite` object, as well as an `update` and a `getPostion` function.

The simple scenario that we've presented here is suitable if we wish to learn about inheritance. Now, let's look at something new: inheriting from the `Soldier` class. Look at the following code, especially the highlighted part:

```
class Sniper : public Soldier
{
public:
    // A constructor specific to Sniper
    Sniper::Sniper();
};
```

By adding `: public Soldier` to the `Sniper` class declaration, `Sniper` inherits from `Soldier`. But what does this mean, exactly? `Sniper` is a `Soldier`. It has all the variables and functions of `Soldier`. Inheritance is even more than this, however.

Also note that, in the previous code, we declare a `Sniper` constructor. This constructor is unique to `Sniper`. We have not only inherited from `Soldier`; we have **extended** `Soldier`. All the functionality (definitions) of the `Soldier` class would be handled by the `Soldier` class, but the definition of the `Sniper` constructor must be handled by the `Sniper` class.

Here is what the hypothetical `Sniper` constructor definition might look like:

```
// In Sniper.cpp
Sniper::Sniper()
{
    setHealth(10);
    setArmour(10);
    setRange(1000);
    setShotPower(100);
}
```

We could go ahead and write a bunch of other classes that are extensions of the `Soldier` class, perhaps `Commando` and `Infantryman`. Each would have the exact same variables and functions, but each could also have a unique constructor that initializes those variables appropriate to the specific type of `Soldier`. `Commando` might have very high `m_Health` and `m_ShotPower` but really puny `m_Range.` `Infantryman` might be in between `Commando` and `Sniper` with mediocre values for each variable.

As if OOP wasn't useful enough already, we can now model real-world objects, including their hierarchies. We can achieve this by subclassing/extending/inheriting from other classes.

The terminology we might like to learn here is that the class that is extended from is the **super-class**, and the class that inherits from the super-class is the **subclass**. We can also say **parent** and **child** class.

You might find yourself asking this question about inheritance: why? The reason is something like this: we can write common code once; in the parent class, we can update that common code, and all the classes that inherit from it are also updated. Furthermore, a subclass only gets to use public and **protected** instance variables and functions. So, designed properly, this also enhances the goals of encapsulation.

Did you say protected? Yes. There is an access specifier for class variables and functions called **protected**. You can think of protected variables as being somewhere between public and private. Here is a quick summary of access specifiers, along with more details about the `protected` specifier:

- **Public** variables and functions can be accessed and used by anyone with an instance of the class.
- **Private** variables and functions can only be accessed/used by the internal code of the class, and not directly from an instance. This is good for encapsulation and when we need to access/change private variables, since we can provide public getter and setter functions (such as `getSprite`). If we extend a class that has private variables and functions, that child class **cannot** directly access the private data of its parent.
- **Protected** variables and functions are almost the same as private ones. They cannot be accessed/used directly by an instance of the class. However, they can be used directly by any class that extends the class they are declared in. So, it is like they are **private**, except for child classes.

To fully understand what protected variables and functions are and how they can be useful, let's look at another OOP topic first. Then, we will see them in action.

Polymorphism

Polymorphism allows us to write code that is less dependent on the types we are trying to manipulate. This can make our code clearer and more efficient. Polymorphism means many forms. If the objects that we code can be more than one type of thing, then we can take advantage of this.

> But what does polymorphism mean to us? Boiled down to its simplest definition, polymorphism means the following: any subclass can be used as part of the code that uses the

superclass. This means we can write code that is simpler and easier to understand and also easier to modify or change. Also, we can write code for the super-class and rely on the fact that no matter how many times it is **subclassed**, within certain parameters, the code will still work.

Let's discuss an example.

Suppose we want to use polymorphism to help write a zoo management game where we must feed and tend to the needs of animals. We will probably want to have a function such as `feed`. We will also probably want to pass an instance of the animal to be fed into the `feed` function.

A zoo, of course, has lots of animals, such as lions, elephants, and three-toed sloths. With our new knowledge of C++ inheritance, it makes sense to code an `Animal` class and have all the different types of animals inherit from it.

If we want to write a function (`feed`) that we can pass `Lion`, `Elephant`, and `ThreeToedSloth` into as a parameter, it might seem like we need to write a `feed` function for each type of `Animal`. However, we can write polymorphic functions with polymorphic return types and arguments. Take a look at the following definition of the hypothetical feed function:

```
void feed(Animal& a)
{
    a.decreaseHunger();
}
```

The preceding function has an `Animal` reference as a parameter, meaning that any object that is built from a class that extends `Animal` can be passed into it.

This means you can write code today and make another subclass in a week, month, or year, and the very same functions and data structures will still work. Also, we can enforce a set of rules upon our subclasses regarding what they can and cannot do, as well as how they do it. So, good design in one stage can influence it at other stages.

But will we ever really want to instantiate an actual animal?

Abstract classes: virtual and pure virtual functions

An **abstract class** is a class that cannot be instantiated and therefore cannot be made into an object.

Some terminology we might like to learn about here is a *concrete* class. A **concrete class** is any class that isn't abstract. In other words, all the classes we have written so far have been concrete classes and can be instantiated into usable objects.

So, it's code that will never be used, then? But that's like paying an architect to design your home and then never building it!

If we, or the designer of a class, want to force its users to inherit from it before using their class, they can make a class **abstract**. If this happens, we cannot make an object from it; therefore, we must inherit from it first and make an object from the subclass.

To do so, we can make a function **pure virtual** and not provide any definition. Then, that function must be **overridden** (**rewritten**) in any class

that inherits from it.

Let's look at an example; it will help. We can make a class abstract by adding a pure virtual function such as the abstract `Animal` class, which can only perform the generic action of `makeNoise`:

```
Class Animal
    private:
        // Private stuff here
public:
        void virtual makeNoise() = 0;
        // More public stuff here
};
```

As you can see, we add the C++ keyword `virtual` before and `= 0` after the function declaration. Now, any class that extends/inherits from `Animal` must override the `makeNoise` function. This might make sense since different types of animals make very different types of noise. We could have assumed that anybody who extends the `Animal` class is smart enough to notice that the `Animal` class cannot make a noise and that they will need to handle it, but what if they don't notice? The point is that by making a pure virtual function, we guarantee that they will because they must, or the code won't compile.

Abstract classes are also useful because, sometimes, we want a class that can be used as a polymorphic type, but we need to guarantee it can never be used as an object. For example, `Animal` doesn't really make sense on its own. We don't talk about animals; we talk about types of animals. We don't say, "Ooh, look at that lovely, fluffy, white animal!" or, "Yesterday, we went to the pet shop and got an animal and an animal bed." It's just too, well, abstract.

So, an abstract class is kind of like a **template** to be used by any class that extends it (inherits from it). If we were building an *Industrial-Empire*-type game where the player manages businesses and their employees, we might want a `Worker` class, for example, and extend it to make `Miner`, `Steelworker`, `OfficeWorker`, and, of course, `Programmer`. But what exactly does a plain `Worker` do? Why would we ever want to instantiate one?

The answer is we wouldn't want to instantiate one, but we might want to use it as a polymorphic type so that we can pass multiple `Worker` subclasses between functions and have data structures that can hold all types of workers.

All pure virtual functions must be overridden by any class that extends the parent class that contains the pure virtual function. This means that the abstract class can provide some of the common functionality that would be available in all its subclasses. For example, the `Worker` class might have the `m_AnnualSalary`, `m_Productivity`, and `m_Age` member variables. It might also have the `getPayCheck` function, which is not pure virtual and is the same in all the subclasses, while a `doWork` function is pure virtual and must be overridden, because all the different types of `Worker` will `doWork` very differently.

By the way, **virtual** as opposed to pure virtual is a function that can be **optionally overridden**. You declare a virtual function the same way as a pure virtual function but leave the `= 0` off to the end. In the current game project, we will use several pure virtual functions.

If any of this virtual, pure virtual, or abstract stuff is unclear, using it is probably the best way to understand it. We will do so soon. First, let's learn

about design patterns and the entity component design pattern.

Design patterns

It is my guess that if you are going to make deep, large-scale games in C++, then design patterns are going to be a big part of your learning agenda in the months and years ahead. What follows will only introduce this vital topic.

A **design pattern** is a reusable solution to a coding problem. In fact, most games (including Run) will use multiple design patterns. The key point about design patterns is this: they are already proven to provide a good solution to a common problem. We are not going to invent any design patterns: we are just going to use some that already exist to solve the problem of our ever-expanding code.

Many design patterns are quite complicated and require further study beyond the level of this book if you want to learn them. What follows is a simplification of a key game development-related pattern. You're urged to continue your study to implement patterns more comprehensively.

Entity Component System pattern

We will now spend five minutes wallowing in the misery of an apparently unsolvable muddle. Then, we will see how the entity-component pattern comes to the rescue.

Why lots of diverse object types are hard to manage

In the previous projects, we coded a class for each object. We had classes such as `Bat`, `Ball`, `Crawler`, and `Zombie`. Then, in the `update` function, we would update them, and in the `draw` function, we would draw them. Each object decides how updating and drawing take place.

We could just get started and use this same structure for Run. It would work, but we are trying to learn something more manageable so that our games can grow in complexity.

Another problem with this approach is that we cannot take advantage of inheritance. For example, all the zombies, the bullets, and the player character from the zombie game draw themselves in an identical way, but unless we change how we do things, we will end up with three `draw` functions with nearly identical code. In the future, if we make a change to how we call the `draw` function or the way we handle graphics, we will need to update all three classes.

There must be a better way.

Using a generic GameObject for better code structure

If every object, player, zombie, and all the bullets were one generic type, then we could pack them away in a `vector` instance and loop through each of their `update` functions, followed by each of their `draw` functions. This is what the main function in the Run project is doing.

We have just learned one way of doing this: inheritance. At first glance, inheritance might seem like a perfect solution. We could create an abstract `GameObject` class and then extend it with the `Player`, `Zombie`, and `Bullet` classes.

The `draw` function, which is identical in all three classes, could remain in the parent class, and we won't have the problem of all that wasted duplicate code. Great!

The problem with this approach is how varied, in some respects, the game objects are. For example, all the object types move differently. The bullets go in a fixed direction, the zombies home in on the player, and the player character responds to keyboard inputs.

How would we put this kind of diversity into the `update` function so that it could control this movement? Maybe we could use something like this:

```
update(){
   switch(objectType){
          case 1:
                // All the player's logic
break;
          case 2:
                // All the zombie's logic here
                Break;
          case 3:
                // All the bullet's logic here
break;
   }
}
```

The `update` function alone would be bigger than the whole `GameEngine` class!

As you may remember from the *Advanced OOP: inheritance and polymorphism section*, when we inherit from a class, we can also override specific functions. This means we could have a different version of the `update` function for each object type. Unfortunately, however, there is also a problem with this approach as well.

The `GameEngine` engine would have to "know" which type of object it was updating or, at the very least, be able to query the `GameObject` instance it was updating in order to call the correct version of the `update` function. What is really needed is for the `GameObject` to somehow internally choose which version of the update `function` is required.

Unfortunately, even the part of the solution that seemed to work falls apart on closer inspection. I said that the code in the `draw` function was the same for all three of the objects, and therefore the `draw` function could be part of the parent class and used by all the subclasses, instead of us having to code three separate `draw` functions. Well, what happens when we introduce a new object that needs to be drawn differently, such as an animated wasp zombie that flies across the top of the screen? In this scenario, the `draw` solution falls apart too.

Now that we have seen the problems that occur when objects are different from each other and yet cry out to be from the same parent class, it is time to look at the solution we will use in the Run project.

What we need is a new way of thinking about constructing all our game objects.

Prefer composition over inheritance

Preferring composition over inheritance refers to the idea of composing objects with other objects.

This concept was first suggested in the following publication:

> *Design Patterns: Elements of Reusable Object-Oriented Software*

—by Erich Gamma, Richard Helm, et al.

What if we could code an entire class (as opposed to a function) that handled how an object was drawn? Then, for all the classes that draw themselves in the same way, we could instantiate one of these special drawing classes within the `GameObject`, and any objects that need to be drawn differently could have a different drawing object. Then, when a `GameObject` does something differently, we simply compose it with a different drawing or update related classes to suit it. All the similarities in all our objects can benefit from using the same code, while all the differences can benefit from not only being encapsulated but also abstracted from (taken out of) the base class.

Note that the heading of this section is composition over inheritance, not composition instead of inheritance. Composition doesn't replace inheritance and everything you learned in the *Advanced OOP: inheritance and polymorphism section*, still holds true. However, where possible, compose instead of inheriting. In the Run project, we will do both.

The `GameObject` class is the entity, while the classes it will be composed of that do things such as update its position and draw it to the screen are the components, which is why it's called the **Entity-Component** pattern.

Have a look at the following diagram, which represents the **Entity-Component** pattern in the form we will implement it in this project:

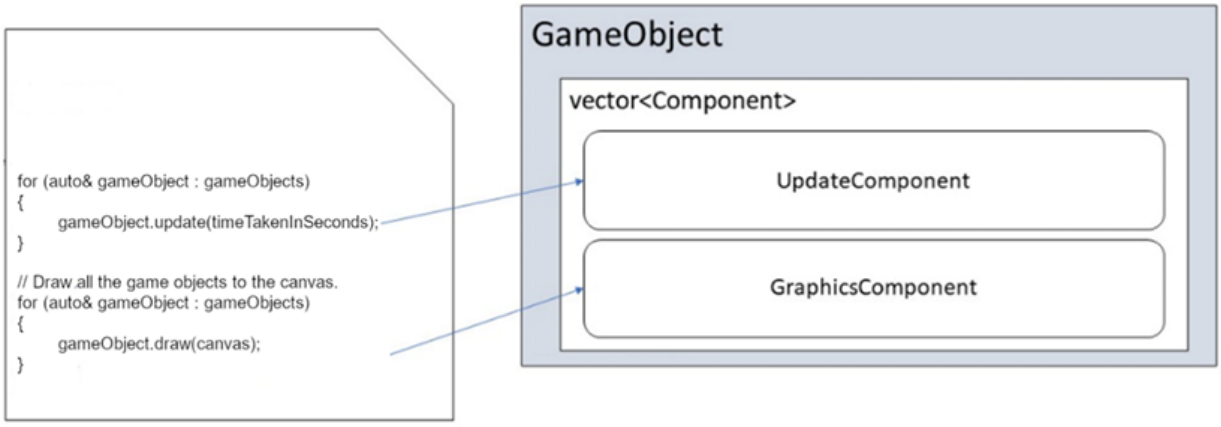


Figure 15.6: Entity component code

In the preceding image, the code on the left is the code from our main function that loops over the `GameObject` vector, first calling `update` and then calling `draw` on each instance in turn. In the preceding diagram, we can see that a `GameObject` instance is composed of multiple `Component` instances. There will be multiple different classes derived from the `Component` class, including `UpdateComponent` and `GraphicsComponent`. Furthermore, there could be further specific classes derived from them. For example, the `BulletUpdateComponent` and `ZombieUpdateComponent` classes could be derived from the `UpdateComponent` class. These classes will handle how an object updates itself in each frame of the game. This is great for encapsulation because we don't need any more big `switch` blocks to distinguish between different objects.

When we use composition over inheritance to create a group of classes that represent behavior/algorithms, as we will here, this is known as the **Strategy** pattern. You could use everything you have learned here and refer to it as the Strategy pattern. Entity-Component is a lesser-known but more specific implementation, and that is why we call it this. The difference is academic, but feel free to turn to ChatGPT if you want to explore things further. A good

resource for further game programming pattern exploration is https://gameprogrammingpatterns.com.

The **Entity-Component** pattern, along with using composition in preference to inheritance, sounds great at first glance but brings with it some problems of its own. It would mean that our new `GameObject` class would need to "know" about all the different types of components and every single type of object in the game. How would it add all the correct components to itself?

Let's have a look at the solution.

Factory pattern

It is true that if we are to have this universal `GameObject` class that can be anything we want it to be, whether that be a bullet, player, invader, or whatever else, then we are going to have to code some logic that "knows" about constructing these super-flexible `GameObject` instances and composes them with the correct components. But adding all this code into the `GameObject` class itself would make it exceptionally unwieldy and defeat the entire reason for using the **Entity-Component** pattern in the first place.

We would need a constructor that did something like this hypothetical `GameObject` code:

```
class GameObject
{
   UpdateComponent* m_UpdateComponent;
   GraphicsComponent* m_GraphicsComponent;
   // More components
// The constructor
GameObject(string type){
      if(type == "invader")
      {
            m_UpdateComp = new InvaderUpdateComponent();
```

```
            m_GraphicsComponent = new StdGraphicsComponent();
        }
        else if(type =="ufo")
         {
                m_UpdateComponent = new
UFOUpdateComponentComponent();
                m_GraphicsComponent = new AnimGraphicsComponent();
         }
        // etc.
        …
    }
};
```

The `GameObject` class would need to know not just which components go with which `GameObject` instance, but also which didn't need certain components, such as input-related components for controlling the player.

For the Run project, we could do this and just about survive the complexity, but for a more complex game, we would likely drown in the code and fail.

The `GameObject` class would also need to understand all this logic. Any benefit or efficiency gained from using composition over inheritance with the Entity-Component pattern would be lost.

Furthermore, what if we decide we want a new type of object, perhaps a new enemy that teleports near to the player, takes a shot, and then teleports away again? It is fine to code a new `GraphicsComponent` class, perhaps a `TeleportGraphicsComponent` that "knows" when it is visible and invisible, along with a new `UpdateComponent`, perhaps a `TeleportUpdateComponent` that teleports instead of moving in a conventional manner, but what is not fine is that we are going to have to add a whole bunch of new `if` statements to the `GameObject` class constructor.

In fact, the situation is even worse than this. What if we decide that regular objects can now teleport? All `GameObjects` now need more than just a different type of `GraphicsComponent` class. We would have to go back into the `GameObject` class to edit all of those `if` statements again.

In fact, there are even more scenarios that can be imagined, and they all end up with a bigger and bigger `GameObject` class. The `Factory` pattern is the solution to these `GameObject` class-related woes and the perfect partner to the Entity-Component pattern.

This implementation of the `Factory` pattern is a big simplification as a way to begin to learn about the `Factory` pattern. Why not do a web search for the `Factory` pattern once you have completed this project and see how it can be improved?

The game designer will provide a specification for each and every type of object in the game, and the programmer will provide a factory class that builds `GameObject` instances from the game designer's specifications. When the game designer comes up with new ideas for entities, then all we need to do is ask for a new specification. Sometimes, that will involve adding a new production line to the factory that uses existing components and, sometimes, it will mean coding new components or perhaps updating existing components. The point is that it won't matter how inventive the game designer is: the `GameObject` and `main` function remain unchanged.

In its simplest form (like our `Factory` class), the `Factory` class has the knowledge to prepare the game objects and their appropriate components for the game loop.

In the `Factory` code, the current object type is instantiated and the appropriate components (classes) are added to it. The fireball, player, platform, and the rest have combinations of different and the same components.

Some game objects will only have graphics like rain and some will only have updates like the level manager, which controls the game's logic.

When we use composition, it can be less clear which class is responsible for the memory. Is it the class that creates it, the class that uses it, or some other class? Let's learn some more C++ to help us manage memory a little more simply.

C++ smart pointers

Smart pointers are classes that we can use to get the same functionality as a regular pointer but with an extra feature: the feature being that they take care of their own deletion. In the limited way we have used pointers so far, it has not been a problem for us to delete our own memory, but as your code becomes more complex, and when you are allocating the new memory in one class but using it in another class, it becomes much less clear which class is responsible for deleting the memory when we are done with it. How can a class or function know whether a different class or function has finished with some allocated memory?

The solution is smart pointers. There are a few types of smart pointers; we will look at two of the most used ones here. The key to success with smart pointers is using the correct type.

The first type we will consider is **shared pointers**.

Shared pointers

The way that a shared pointer can safely delete the memory it points to is by keeping a count of the number of different references there are to an area of memory. If you pass a pointer to a function, the count is increased by one. If you pack a pointer into a vector, the count is increased by one. If the function returns, the count is decreased by one. If the vector goes out of scope or has the `clear` function called on it, the smart pointer will reduce the reference count by one. When the reference count is zero, nothing points to the area of memory anymore and the smart pointer class calls `delete`. All the smart pointer classes are implemented using regular pointers behind the scenes. We just get the benefit of not having to concern ourselves about where or when to call `delete`. Let's look at the code for using a shared smart pointer.

The following code creates a new shared smart pointer called `myPointer` that will point to an instance of `MyClass`:

```
shared_ptr<MyClass> myPointer;
```

`shared_ptr<MyClass>` is the type while `myPointer` is its name. The following code is how we might initialize `myPointer`:

```
myPointer = make_shared<MyClass>();
```

The call to `make_shared` internally calls `new` to allocate the memory. The parentheses `()` is the constructor parentheses. If the `MyClass` class constructor took an `int` parameter, for example, the preceding code might look like this:

```
myPointer = make_shared<MyClass>(3);
```

The `3` in the preceding code is an arbitrary example.

Of course, you can declare and initialize your shared smart pointers in a single line of code if required, as shown in the following code:

```
shared_ptr<MyClass> myPointer = make_shared<MyClass>();
```

It is because `myPointer` is a `shared_ptr` that it has an internal reference count that keeps track of how many references point to the area of memory it created. If we make a copy of the pointer, that reference count is increased.

Making a copy of the pointer includes passing the pointer to another function, placing it in a `vector`, `map`, or other structure, or simply copying it.

We can use a smart pointer using the same syntax as a regular pointer. It is quite easy to forget sometimes that it isn't a regular pointer. The following code calls the `myFunction` function on `myPointer`:

```
myPointer->myFunction();
```

By using a shared smart pointer, there is some performance and memory `overhead`. By overhead, I mean that our code runs slower and uses more memory. After all, the smart pointer needs a variable to keep track of the reference count, and it must check the value of the reference count every time a reference goes out of scope. However, this overhead is tiny and only an issue in the most extreme situations since most of the overhead happens while the smart pointers are being created. Typically, we will create smart pointers outside of the game loop. Calling a function on a smart pointer is as efficient as a regular pointer.

Sometimes, we know that we will only ever want one reference to a smart pointer, and in this situation, **unique pointers** are the best option.

Unique pointers

When we know that we only want a single reference to an area of memory, we can use a **unique smart pointer**. Unique pointers lose much of the overhead that I mentioned shared pointers have. In addition, if you try and make a copy of a unique pointer, the compiler will warn us, and the code will either not compile or it will crash, giving us a clear error. This is a very useful feature that can prevent us from accidentally copying a pointer that was not meant to be copied. You might be wondering if this no-copying rule means we can never pass it to a function or even put it in a data structure such as a `vector`. To find out, let's look at some code for unique smart pointers and explore how they work.

The following code creates a unique smart pointer called `myPointer` that points to an instance of `MyClass`:

```
unique_ptr<MyClass> myPointer = make_unique<MyClass>();
```

Now, let's suppose we want to add a `unique_ptr` to a `vector`. The first thing to note is that `vector` must be of the correct type. The following code declares a `vector` that holds unique pointers to `MyClass` instances:

```
vector<unique_ptr<MyClass>> myVector;
```

The `vector` is called `myVector` and anything you put into it must be of the unique pointer type to `MyClass`. But didn't I say that unique pointers can't be copied? When we know that we will only ever want a single reference to an

area of memory, we should use `unique_ptr`. This doesn't mean, however, that the reference can't be moved. Here is an example:

```
// Use move() because otherwise
// the vector has a COPY which is not allowed
mVector.push_back(move(myPointer));
// mVector.push_back(myPointer); // Won't compile!
```

In the preceding code, we can see that the `move` function can be used to put a unique smart pointer into a `vector`. Note that when you use the `move` function, you are not giving the compiler permission to break the rules and copy a unique pointer: you are moving responsibility from the `myPointer` variable to the `myVector` instance. If you attempt to use the `myPointer` variable after this point, the code will execute and the game will crash, giving you a **Null pointer access violation error message**. The following code will cause a crash:

```
unique_ptr<MyClass> myPointer = make_unique<MyClass>();
vector<unique_ptr<MyClass>> myVector;
// Use move() because otherwise
// the vector has a COPY which is not allowed
mVector.push_back(move(myPointer));
// mVector.push_back(myPointer); // Won't compile!
myPointer->myFunction();// CRASH!!
```

The exact same rules apply when passing a unique pointer to a function; use the `move` function to pass responsibility on. We will look at some of these scenarios again, as well as some more when we get back to the Run project in a while.

Casting smart pointers

We will often want to pack the smart pointers of derived classes into data structures or function parameters of the base class such as all the different derived `Component` classes. This is the essence of polymorphism. Smart pointers can achieve this using casting. But what happens when we later need to access the functionality or data of the derived class?

A good example of where this will regularly be necessary is when we deal with components inside our game objects. There will be an abstract `Component` class and derived from that there will be `GraphicsComponent`, `UpdateComponent`, and more.

As an example, we will want to pass generic component-based classes to functions and yet use the functions of the derived classes. But if all the components are stored as base class `Component` instances, then it might seem that we can't do this. Casting from the base class to a derived class solves this problem.

The following code casts `myComponent`, which is a base class `Component` instance, to an `UpdateComponent` class instance, which we can then call the `update` function on:

```
shared_ptr<UpdateComponent> myUpdateComponent =
                static_pointer_cast<UpdateComponent>
(MyComponent);
```

Before the equals sign, a new `shared_ptr` to an `UpdateComponent` instance is declared. After the equals sign, the `static_pointer_cast` function specifies the type to cast to in the angle brackets, `<UpdateComponent>`, and the instance to cast from in parentheses, `(MyComponent)`.

We can now use all the functions of the `UpdateComponent` class, which in our project includes the `update` function. We would call the `update` function as

follows:

```
myUpdateComponent->update(fps);
```

There are two ways we can cast a class smart pointer to another class smart pointer. One is by using `static_pointer_cast`, as we have just seen, and the other is to use `dynamic_pointer_cast`. The difference is that `dynamic_pointer_cast` can be used if you are uncertain whether the cast will work. When you use `dynamic_pointer_cast`, you can then check to see whether it worked by testing whether the result is a null pointer. You use `static_pointer_class` when you are certain the result is the type you are casting to. We will use `static_pointer_cast` in a couple of places in the Run project. Let's get back to building our game.

Coding the GameObject class

The `GameObject` class depends on the `Component` class and the `Component` class depends on the `Graphics` and `Update` classes, so, let's code all four.

Remember that in our discussion about the entity component system, we talked about `Component` classes and that `GraphicsComponent`, `UpdateComponent`, and so on would derive from `Component`. For the purposes of presenting the code, we will shorten `GraphicsComponent` to just `Graphics` and `UpdateComponent` to just `Update`.

Create a class called `GameObject`. In `GameObject.h`, add the following code:

```
#pragma once
#include "SFML/Graphics.hpp"
#include "Component.h"
#include <vector>
using namespace sf;
```

```
using namespace std;
class GameObject
{
private:
  vector <shared_ptr<Component>>  m_Components;
public:
  void addComponent(shared_ptr<Component> newComponent);
  void update(float elapsedTime);
  void draw(VertexArray& canvas);
};
```

In the code, there are errors because we are missing our `Component` class. As you have probably come to expect, we will code that shortly.

In the preceding code, we have a vector for holding `Component` instances. We will not add any abstract `Component` instances into the vector but derived `Graphics` and `Update` instances. To facilitate this, we have the `addComponent` function.

We also have an `update` and a `draw` function. We have already seen these two functions being called from the main game loop. Here is the code from the game loop in the `main` function as a refresher. You do not need to add this code again:

```
// Update all the game objects.
for (auto& gameObject : gameObjects)
{
  gameObject.update(timeTakenInSeconds);
}
// Draw all the game objects to the canvas.
for (auto& gameObject : gameObjects)
{
  gameObject.draw(canvas);
}
```

Hopefully, you can see how this whole system is coming together.

Let's code the three functions of the `GameObject` class. In `GameObject.cpp` add the following code:

```
#include "GameObject.h"
#include "SFML/Graphics.hpp"
#include <iostream>
#include "Update.h"
#include "Graphics.h"
using namespace std;
using namespace sf;
void GameObject::addComponent(
  shared_ptr<Component> newComponent)
{
  m_Components.push_back(newComponent);
}
void GameObject::update(float elapsedTime)
{
  for (auto component : m_Components)
  {
    if (component->m_IsUpdate)
    {
      static_pointer_cast<Update>
        (component)->update(elapsedTime);
    }
  }
}
void GameObject::draw(VertexArray& canvas)
{
  for (auto component : m_Components)
  {
    if (component->m_IsGraphics)
    {
      static_pointer_cast<Graphics>
        (component)->draw(canvas);
    }
  }
}
```

In this file, there are three missing classes that are causing errors. They are `Component`, `Update`, and `Graphics`. `Update` and `Graphics` will be derived from `Component` and we will code them all after we have discussed the code we have just added.

In the preceding code, the `addComponent` function has just one line of code that uses the `push_back` function of `vector` to add a new instance of a derived component into the `m_Components` vector.

The `update` function is short and simple, too. First, the code loops through all the components like this:

```
for (auto component : m_Components)
{
```

Then, it checks whether the current component is an update component like this:

```
if (component->m_IsUpdate)
  {
```

Finally, if the preceding test is true, the `update` function is called, and the instance will execute its own version of the `update` function. Remember that this could be anything from our game: the player, a fireball, a menu, and so on: anything.

The `draw` function does exactly the same as the `update` function, except that it looks for a graphics component and calls the `draw` function.

The preceding code implies that the `Component` class will have the Boolean variables `m_IsUpdate` and `m_IsGraphics`. Let's code the `Component` class next.

Coding the Component class

The `Component` class is the shortest class in the book. It has no functions. It just exists to be extended. In fact, we will leave the `Component.cpp` file empty. Note, however, that we are expanding very slightly upon the simple entity component example from earlier. `Graphics` and `Update` will extend `Component`. `Component` will be the **polymorphic** type, but `Graphics` and `Update` will be the **abstract** classes (with pure virtual functions) that all the useable classes of our game will extend. Create a class called `Component` and in `Component.h`, add the following code:

```
#pragma once
#include <iostream>
using namespace std;
class Component
{
public:
  bool m_IsGraphics = false;
  bool m_IsUpdate = false;
};
```

In the preceding code, we create a class called `Component` and add two public member variables. The `m_IsGraphics` and `m_IsUpdate` Booleans will be set when a new component is added and tested before updating or drawing. That's it.

`Component.cpp` will remain empty because there is no functionality. You could delete `Component.cpp` if you wish.

There is, however, much more to the classes that extend `Component`. Let's code the `Graphics` class first and then we will move on to the `Update` class.

Coding the Graphics class

We will call this class, which derives from `Component`, `Graphics`. The next class, which also derives from `Component`, we will call `Update`. It would be more apparent and clearer if we called them `GraphicsComponent` and `UpdateComponent` but component is a long word. Therefore, I opted simply for `Graphics` and `Update`. I may from time to time refer to `Graphics` and `Update` as components because they are, even if not by name.

Create a class called `Graphics` that has `Component` as its base class. You could add `Component` in the **Base class** field of the **New class** dialog box and a little bit more code will be auto-generated for you. But simply coding the following to `Graphics.h` will also have the exact same effect.

In `Graphics.h`, add the following code:

```
#pragma once
#include "Component.h"
#include <SFML/Graphics.hpp>
using namespace sf;
class Update;
class Graphics :
    public Component
{
private:
public:
    Graphics();
    virtual void assemble(
        VertexArray& canvas,
        shared_ptr<Update> genericUpdate,
        IntRect texCoords) = 0;
    virtual void draw(VertexArray& canvas) = 0;
};
```

In the preceding code, there are no variables and just two public functions. Look carefully at the functions; they have the tell-tale `virtual` at the start and `= 0` at the end of the declarations. Any class that extends this class must implement (provide a definition for) the two functions. The first pure virtual function of the `Graphics` interface is the `assemble` function. As we proceed, we will write a whole bunch of classes that extend the `Graphics` class, including `PlayerGraphics`, `RainGraphics`, and `PlatformGraphics`. Each will provide its own specific implementation of the `assemble` function. This is useful because they will all need to be assembled in a slightly different way.

Before we move on, note the signature of the `assemble` function. First, there is a `VertexArray` reference, which will allow the addition of texture coordinates for the required graphic. There is a shared pointer to an `Update` instance. We will see how we can use this to get the required data from the `Update` instance, which corresponds to this current `Graphics` instance. We will use static casting, as discussed in the *Casting smart pointers* section, to access the functions of the appropriate child class.

Finally, we have an SFML `IntRect` instance, which will contain the texture coordinates for this object. The `assemble` function will be called in the `loadLevel` function of the `Factory` class.

The `draw` function receives the `VertexArray` during iteration through the main game loop, allowing it to update its position.

In `Graphics.cpp`, add the following code:

```
#include "Graphics.h"
Graphics::Graphics()
{
  m_IsGraphics = true;
}
```

In the preceding code, the constructor does just one thing. It sets the `m_IsGraphics` Boolean to `true`. When any instance that derives from `Graphics` is created, the compiler will always call this constructor, which guarantees that the public variable declared in the `Component` class is set appropriately. Remember that this value is checked in the `GameObject` code before attempting to call the `draw` function.

Coding the Update class

Create a class called `Update` that has `Component` as its base class. Use the **Base class** field if you prefer, or don't: either way is fine.

In `Update.h`, add the following code:

```
#pragma once
#include "Component.h"
#include "SFML/Graphics.hpp"
class LevelUpdate;
class PlayerUpdate;
class Update :
    public Component
{
private:
public:
    Update();
    virtual void assemble(
        shared_ptr<LevelUpdate> levelUpdate,
        shared_ptr<PlayerUpdate> playerUpdate) = 0;
    virtual void update(float timeSinceLastUpdate) = 0;
};
```

In the preceding code, we have two pure virtual functions: they are `assemble` and `update`. The `assemble` function will be used by the `Factory` class in the `loadLevel` function. We can see from the signature that the `assemble`

function uses two shared pointers: one is a `LevelUpdate` instance and the other is a `PlayerUpdate` instance. We haven't coded these yet, but all `Update`-derived instances will need to keep track of the game state, which will be controlled by the `LevelUpdate` class (also an `Update`-derived class) and also the state of the player controlled by the `PlayerUpdate` class.

We get away with referring to these two classes that don't exist yet by adding the forward declarations near the start of `Update.h`, as shown again next:

```
class LevelUpdate;
class PlayerUpdate;
```

If we attempted to use either of those shared pointers before the classes are implemented, the code wouldn't work, but just adding them in a function signature works fine because of the forward declarations.

In `Update.cpp`, add the following code:

```
#include "Update.h"
Update::Update()
{
  m_IsUpdate = true;
}
```

In the preceding code, we use the same technique as in the `Graphics` class to make sure the parent `Component` class sets the appropriate Boolean so the `GameObject` class can know what type of component (`Graphics` or `Update`) it is currently using.

Running the code

Now that all the code is error free, we can run it, but we just get a grey screen. Furthermore, we can't easily stop the program. Use the *Ctrl* + *Alt* + *Delete* keyboard combination, select `Run.exe`, and then press **End Task** to force stop the program.

To add some temporary code to fix this inconvenience, find this code in `InputDispatcher.cpp`:

```
//if (event.type == Event::KeyPressed &&
//    event.key.code == Keyboard::Escape)
//{
//    m_Window->close();
//}
```

Uncomment the preceding code and it will allow the `InputDispatcher` instance to handle when the *Esc* key is pressed. The `InputDispatcher` class should only handle dispatching input messages, but we will cheat until we implement our menu-related classes later in the project. You can now run the program, admire the grey screen, and conveniently press *Esc* to quit.

What next?

We need to talk about the new way graphics will work in this project and we will do so in detail in *Chapter 17*. When I say new, I mean new to this book, as it is a technique that goes back to the beginning years of game development.

If you look in the `graphics` folder, there is just one graphic. Furthermore, we have not called the `window.draw` function at any point in our code so far. We will discuss why draw calls should be kept to a minimum as well as implement our camera-related classes that will handle this for us.

The reason we will defer this discussion is it helps to have some working code to discuss. Of course, vertex arrays and texture coordinates are not new to us as we used them for the background in the zombie project. Therefore, in the next chapter, we will begin to implement the game logic and the first part of the player-related classes. Furthermore, as it will be easy because we have handled sound before, we will implement a `SoundManager` class with the added ability to play a short tune on a loop.

Now, let's have a refresher on everything we have done and learned in this chapter.

Summary

First, we looked at exactly what the new game is and how it will be played. Then, we made the project in the usual way and coded the shortest `main` function (main game loop) of the entire book!

Next, we began coding the new way we will handle the player's input by delegating specific responsibilities to individual game entities/objects and having them listen for messages from a new `InputDispatcher` class.

We coded a class called `Factory` that will be responsible for "knowing" how to assemble all the different components we will build into usable derived types before being placed/composed inside `GameObject` instances.

We learned about C++ **inheritance**, **polymorphism**, and C++ **smart pointers** for passing responsibility for memory management to the compiler.

Then, we coded the key `GameObject` class. The `Component` class, which is the parent class for almost every other class, we will code in the rest of the book and which the `GameObject` instances will hold. Next, we coded the `Graphics`

and `Update` classes, which will be derived/extended from/children of `Component`.

We are all set to add sound and game logic and learn about inter-object communication in the next chapter.

16

Sound, Game Logic, Inter-Object Communication, and the Player

In this chapter, we will quickly implement our game's sound. We have done this before, so it won't be hard. In fact, in just half a dozen lines of code, we will also add music playing to our sound features. Later in the project (but not in this chapter), we will add **directional** (spatialized) sound. This time, however, we will wrap all our sound-related code into a single class called `SoundEngine`. Once we have some noise, we will then move on to get started on the player. We will achieve the entire player character functionality just by adding two classes: one that extends `Update` and one that extends `Graphics`. This creation of new game objects by extending these two classes will be how we do almost everything else for the entire game. We will also see the simple way that objects communicate with each other using pointers. The completed code for this chapter can be found in the `Run2` folder.

In a nutshell, in this chapter, we will:

- Code the `SoundEngine` class: Code a sound-related class that also plays music in a loop

- Code the Game logic: Code a class that handles all the game logic and learn how it will communicate with all the other game objects.
- Coding the player: Code the first part of our player using a graphics component and an update component.
- Coding the factory to use all our new classes: Code some more of the factory, which knows how to assemble different game objects and share the appropriate data between them.
- Running the game.

We'll start by adding a sound class.

Coding the SoundEngine class

You might recall from the previous project that all the sound code took up quite a few lines. Now, consider that we will need even more code when we add spatialization in *Chapter 20*; it's going to get even longer. To keep our code manageable, we will code a class to manage all our sound effects and music being played.

All this code will be very familiar. Even the new feature of playing some music should seem quite intuitive because of what we did in the other games. Create a new class called `SoundEngine`. In the `SoundEngine.h` file, add the following code:

```
#pragma once
#include <SFML/Audio.hpp>
using namespace sf;
class SoundEngine
{
private:
  static Music music;
  static SoundBuffer m_ClickBuffer;
```

```
    static Sound m_ClickSound;
    static SoundBuffer m_JumpBuffer;
    static Sound m_JumpSound;
  public:
    SoundEngine();
    static SoundEngine* m_s_Instance;
    static bool mMusicIsPlaying;
    static void startMusic();
    static void pauseMusic();
    static void resumeMusic();
    static void stopMusic();
    static void playClick();
    static void playJump();
  };
```

In the preceding code, we have an SFML `Music` object, `SoundBuffer`, and a `Sound` object for each sound effect we are going to play. In the `public` section of the class, we have functions to start, pause, stop, and resume the music, along with two functions to play each of the sound effects. It should be trivial to add as many sound effects as you like to the game once we see how this works.

In the `SoundEngine.cpp` class, add the following code:

```
#include "SoundEngine.h"
#include <assert.h>
SoundEngine* SoundEngine::m_s_Instance = nullptr;
bool SoundEngine::mMusicIsPlaying = false;
Music SoundEngine::music;
SoundBuffer SoundEngine::m_ClickBuffer;
Sound SoundEngine::m_ClickSound;
SoundBuffer SoundEngine::m_JumpBuffer;
Sound SoundEngine::m_JumpSound;
SoundEngine::SoundEngine()
{
  assert(m_s_Instance == nullptr);
  m_s_Instance = this;
  m_ClickBuffer.loadFromFile("sound/click.wav");
```

```
    m_ClickSound.setBuffer(m_ClickBuffer);
    m_JumpBuffer.loadFromFile("sound/jump.wav");
    m_JumpSound.setBuffer(m_JumpBuffer);
}
void SoundEngine::playClick()
{
    m_ClickSound.play();
}
void SoundEngine::playJump()
{
    m_JumpSound.play();
}
void SoundEngine::startMusic()
{
    music.openFromFile("music/music.wav");
    m_s_Instance->music.play();
    m_s_Instance->music.setLoop(true);
    mMusicIsPlaying = true;
}
void SoundEngine::pauseMusic()
{
    m_s_Instance->music.pause();
    mMusicIsPlaying = false;
}
void SoundEngine::resumeMusic()
{
    m_s_Instance->music.play();
    mMusicIsPlaying = true;
}
void SoundEngine::stopMusic()
{
    m_s_Instance->music.stop();
    mMusicIsPlaying = false;
}
```

The sound effects are implemented as we did in the previous projects except, now, we have encapsulated them in a class. The buffers and sounds are loaded and associated in the constructor and the related function calls play on the appropriate `Sound` instance.

Let's explore how the music works. `Music` instances do not have buffers. Technically speaking, you could load a music file in a regular `Sound` object, but as music is usually much longer than a sound effect, this wouldn't give good results. Therefore, SFML provides the `Music` class. In the `startMusic` function, you can see that we use the `openFromFile` function. This prepares the file to be streamed rather than loaded all at once. Then, we call the `music.play` function, which begins streaming and plays the music. Next, we call `music.setLoop` and pass in `true`. This makes the music repeat over and over.

In the `pauseMusic`, `resumeMusic`, and `stopMusic` functions, we call the SFML-provided `pause`, `play`, and `stop` functions, respectively. Notice we also set the `m_MusicIsPLaying` Boolean appropriately so we can keep track of the state of the music.

We have some more code to add to the sound manager toward the end of the project when we add directional sound, so we can hear if the fireballs are coming from the left or the right.

Code the Game logic

To control the game logic, we will encapsulate it in a game object right in the thick of the game and provide the necessary communication connections out to other game objects and inward from other game objects. This communication will be in the form of pointers to key values. For example, all objects will have a pointer to the logic-related game object to know such things as when the game is paused, among other things.

The idea of putting the game logic in a separate class is interesting. Consider a scenario if your game should have three different game modes.

Imagine the confusing mess of `if`, `else`, and `else if` statements that would be required if we incorporated all that logic into the main function. This way, the factory can simply pick a game object based on the game mode the player chooses. While this game will only have a single game mode, once you see the code, creating a different set of logic in a different class will be trivial.

Note that there won't be a `LevelGraphics` class because we don't need one. Later in the project, when we create a rain effect game object, we will see there will be a `RainGraphics` class that extends `Graphics` but there will be no requirement for an `Update` derived object. Most game objects we create will have an update and a graphics-based component. The point is it is a flexible system.

Coding the LevelUpdate class

Create a new class called `LevelUpdate` that uses `Update` as a **base** class. Add the following code to `LevelUpdate.h`:

```
#pragma once
#include "Update.h"
using namespace sf;
using namespace std;
class LevelUpdate : public Update
{
private:
  bool m_IsPaused = true;
  vector <FloatRect*> m_PlatformPositions;
  float* m_CameraTime = new float;
  FloatRect* m_PlayerPosition;
  float m_PlatformCreationInterval = 0;
  float m_TimeSinceLastPlatform = 0;
  int m_NextPlatformToMove = 0;
  int m_NumberOfPlatforms = 0;
```

```
    int m_MoveRelativeToPlatform = 0;
    bool m_GameOver = true;
    void positionLevelAtStart();
  public:
    void addPlatformPosition(FloatRect* newPosition);
    void connectToCameraTime(float* cameraTime);
    bool* getIsPausedPointer();
    int getRandomNumber(int minHeight, int maxHeight);
    // From Update : Component
  void update(float fps) override;
    void assemble(
      shared_ptr<LevelUpdate> levelUpdate,
      shared_ptr<PlayerUpdate> playerUpdate)
   override;
  };
```

In the preceding code, there are a lot of member variables, as follows:

- The `m_IsPaused` Boolean simply keeps track of whether the game is paused or not. Any game object that needs to know this will obtain a pointer to this value.
- The `m_PlatformPositions` variable is a vector that holds pointers to `FloatRect` instances. As the name suggests, these instances will hold the position and size of all the platforms in the game. It is important to know that the pointers, once initialized, will point directly to the values within the platform-related game objects. This means that this class can directly manipulate the platforms. We will see how this is achieved when we code the platforms. We will see how we manipulate these platform positions when we code the functions of this class.
- The `m_CameraTime` variable is a simple float. It holds the number of seconds, including fractions of a second, that the current attempt at the game has been running for. This is the key benchmark of success for

the player. Soon, we will display this on the screen in the top-left corner.

- The `m_PlayerPosition` pointer is a pointer to a `FloatRect` instance that holds the position of the player. As it will point directly into the player-related class, the `LevelUpdate` class will be able to make decisions based on the current location of the player, like whether the player has fallen too far behind and the game is over.
- The `m_PlatformCreationInterval` float variable will hold the amount of time to wait between creating new platforms. As we will soon see, we don't create new platforms, we just reuse a set of platforms. The interval will be based on the length of the previously reused/new platform. This makes sense because the player will have further to run on some platforms, and making the time interval relative to the platform length seems fair.
- The `m_TimeSinceLastPlatform` float variable works in conjunction with `m_PlatformCreationInterval`. When `m_TimeSinceLastPlatform` is equal to or greater than `m_PlatformCreationInterval`, then it is time to create/reuse another platform in front of the preceding platform.
- The `m_NextPlatformToMove` int variable will represent the position in the vector of platform positions of the next platform that will be reused.
- The `m_NumberOfPlatforms` int variable is the number of platforms that have been created. The code works with a very low number, such as 5, or a much higher number, such as 500. The main difference is that the smallest, most efficient number that makes the game playable and the code efficient is what we will use in the `Factory` class's `loadLevel` function.

- The `m_MoveRelativeToPlatform` int variable is the position in the vector of platform positions that the next platform will be moved relative to. Think about running along the newest platform with nowhere left to go and then the next platform spawns just in time. That next platform needs to be in an accessible position relative to the previous one.
- The `m_GameOver` Boolean keeps track of whether the game has ended or, when the program is first executed, that the game has not started yet. This is distinct from `m_IsPaused`.

Now let's learn about the functions:

- The first is a **private** function called `positionLevelAtStart`, which sets up the initial position of all the game objects at the start of each game.
- The `addPlatformPosition` function receives a `FloatRect` pointer called `newPosition` and will position individual platforms.
- The `connectToCameraTime` function receives a float pointer that can be kept in synchronization with `m_CameraTime`. It is through this mechanism that we will update the text on the screen that displays the time to the player. The text will be drawn using an SFML Text instance in the `CameraGraphics` class, which we will code in the next chapter.
- The `getIsPausedPointer` function returns a pointer to a Boolean. Specifically, it returns a pointer to the `m_IsPaused variable`. This allows access to whether the game is paused to any part of our code that needs it. We will see this in action throughout the rest of the project, as multiple game objects need to know if the game is paused.
- The `getRandomNumber` function takes two values and returns a random number in between. We will see code like this throughout the project.

The most common use for this function in this class is determining where to position platforms when they are reused.

Finally, we have the two overridden functions that are inherited from the `Update` class:

- The `update` function receives the time duration that the last loop of the game took to execute. Just as with our other games, this will be crucial for timing all the actions in the `update` function.
- The `assemble` function, as previously described, will be used in the factory for preparing the component for use. Once we have finished coding the `LevelUpdate` class, we will code some player-related classes. Then, we will get to see how we use assemble from within the `Factory` class.

Next, we will add the code to the `LevelUpdate.cpp` file. As we saw from the `LevelUpdate.h` file, there are quite a few functions. Therefore, we will add and explain the functions in a few parts. Add the following code to the `LevelUpdate.cpp` file to get started:

```
#include "LevelUpdate.h"
#include<Random>
#include "SoundEngine.h"
#include "PlayerUpdate.h"
using namespace std;
void LevelUpdate::assemble(
  shared_ptr<LevelUpdate> levelUpdate,
  shared_ptr<PlayerUpdate> playerUpdate)
{
  m_PlayerPosition = playerUpdate->getPositionPointer();
  //temp
  SoundEngine::startMusic();
}
void LevelUpdate::connectToCameraTime(float* cameraTime)
{
```

```
    m_CameraTime = cameraTime;
}
void LevelUpdate::addPlatformPosition(FloatRect* newPosition)
{
    m_PlatformPositions.push_back(newPosition);
    m_NumberOfPlatforms++;
}
bool* LevelUpdate::getIsPausedPointer()
{
    return &m_IsPaused;
}
```

In the preceding code, we add the required include directives. Notice there are errors among the include directives and the functions because we haven't coded the player yet.

In the `assemble` function, we call `playerUpdate->getPositionPointer`. This implies that when we create the `PlayerUpdate` class, we will also create a function called `getPositionPointer`. We will do this soon, but the point to note now is that the `LevelUpdate` instance will always be able to see the position of the player. Next, as a temporary measure, we call `startMusic` so we can hear music for the first time. Eventually, a menu-related game object will control starting, stopping, and pausing the music.

In the `connectToCameraTime` function, we initialize `m_CameraTime` with the memory address contained in `cameraTime`. We won't actually call this function for a while, but it is ready for when we need it.

The `addPlatformPosition` function uses `push_back` to add the passed-in platform position into the vector. Each time we create a new platform in the factory, we will call this function. We also increment the `m_NumberOfPlatforms` variable to keep track of how many platforms we have.

The `getPausedPointer` function returns the address of the `m_IsPaused` Boolean providing permanent access to the game's state to anything that requests and keeps the returned address.

Next, add the `positionLevelAtStart` function to the `LevelUpdate.cpp` file:

```
Void LevelUpdate::positionLevelAtStart()
{
  float startOffset = m_PlatformPositions[0]->left;
  for (int I = 0; i < m_NumberOfPlatforms; ++i)
  {
    m_PlatformPositions[i]->left = i * 100 + startOffset;
    m_PlatformPositions[i]->top = 0;
    m_PlatformPositions[i]->width = 100;
    m_PlatformPositions[i]->height = 20;
  }
  m_PlayerPosition->left =
    m_PlatformPositions[m_NumberOfPlatforms / 2]->left + 2;
  m_PlayerPosition->top =
    m_PlatformPositions[m_NumberOfPlatforms / 2]->top-- 22;
  m_MoveRelativeToPlatform = m_NumberOfPlatforms-- 1;
  m_NextPlatformToMove = 0;
}
```

In the `positionLevelAtStart` function, the first line of code initializes a `float` variable called `startOffset` by getting the left-hand coordinate of the first platform in the vector. Next, the code loops through all the platforms in the vector from zero through `m_NumberOfPlatforms`. Each iteration of the loop positions a platform at `i * 100` hundred units `+` start offset horizontally, zero units vertically, 100 units in width, and 20 units in height. This could have been done in the factory but when the player started the second playthrough, the platforms will likely be all over the place. The result is that all the platforms are lined up end to end in a straight line with

no variation in size or height. This is like an easy start for the player before positions start to get randomized.

Outside the `for` loop, using the next two lines, the player is positioned to the left-hand edge of the approximately middle platform in the vector by using `[m_NumberOfPlatforms / 2]`. The magic numbers `+2` horizontally and `- 22` vertically are used to make sure the player's feet are firmly on this platform. You are invited to improve this code once you have seen how we code the `Factory` class.

The next line of code initializes the `m_MoveRelativeToPlatform` to the final platform in the vector. This makes sense because we want to keep placing new platforms beyond the right-hand edge of the farthest right platform. The final line of code sets the first platform in the vector as the next candidate to be moved. This means that the platform on the farthest left will be moved to the farthest right and the player will spawn in the middle.

Next, add the `getRandomNumber` function. This function does exactly as the name suggests and we will use it in a few places throughout the code when we want to generate a random value between the two values that we pass into it. Add the code that follows to the `LevelUpdate.cpp` file:

```
int LevelUpdate::getRandomNumber(int minHeight, int maxHeight)
{
#include <random>
// Seed the random number generator with current time
  random_device rd;
  mt19937 gen(rd());
  // Define a uniform distribution for the desired range
uniform_int_distribution<int>
 distribution(minHeight, maxHeight);
  // Generate a random height within the specified range
int randomHeight = distribution(gen);
```

```
    return randomHeight;
}
```

This function is a more modern way to generate a random number than we used in the previous games.

The first line creates a random device object `rd`, which is used to seed the random number generator. The `random_device` is a source of non-deterministic random numbers, often based on hardware values. This is much more reliable than the previous methods we used.

Next, a **Mersenne Twister pseudo-random number generator** (`mt19937`) is initialized with the random device's seed (`rd`). The Mersenne Twister is a widely used algorithm for generating very high-quality random numbers.

Next, a `uniform_int_distribution` instance called `distribution` creates a uniform distribution object for generating integers in the specified range (`minHeight` to `maxHeight`, inclusive). The `uniform_int_distribution` class ensures that each integer in the range has an equal probability of being selected.

The `distribution(gen)` code generates a random integer using the previously defined distribution and the Mersenne Twister generator. The result is stored in the `randomHeight variable`.

Finally, the randomly generated number is returned to the calling code. All you need to remember is that if you call this function, you will get a genuinely random value somewhere in between the two values passed in.

The last function for the `LevelUpdate` class is the `update` function. Recall that the `update` function is called every frame by the `GameObject` class, which, in turn, is called every frame by the game loop. This is a relatively

complex function that handles the entire game logic. Try and study the structure as you add it, and we will then discuss how it works.

Add the `update` function to the `LevelUpdate` class. We will break this code down and talk about it, but I recommend copying and pasting or coding the entire function in one go as it would be very easy to get the structure mixed up when coding it in sections:

```
void LevelUpdate::update(float timeSinceLastUpdate)
{
  if (!m_IsPaused)
  {
    if (m_GameOver)
    {
      m_GameOver = false;
      *m_CameraTime = 0;
      m_TimeSinceLastPlatform = 0;
      int platformToPlacePlayerOn;
      positionLevelAtStart();
    }
    *m_CameraTime += timeSinceLastUpdate;
    m_TimeSinceLastPlatform += timeSinceLastUpdate;
    if (m_TimeSinceLastPlatform > m_PlatformCreationInterval)
    {
      m_PlatformPositions[m_NextPlatformToMove]->top =
        m_PlatformPositions[m_MoveRelativeToPlatform]->top +
        getRandomNumber(-40, 40);
      // How far away to create the next platform
// Bigger gap if lower than previous
if (m_PlatformPositions[m_MoveRelativeToPlatform]->top
        < m_PlatformPositions[m_NextPlatformToMove]->top)
      {
        m_PlatformPositions[m_NextPlatformToMove]->left =
          m_PlatformPositions[m_MoveRelativeToPlatform]-
>left +
          m_PlatformPositions[m_MoveRelativeToPlatform]-
>width +
          getRandomNumber(20, 40);
      }
```

```
      else
      {
        m_PlatformPositions[m_NextPlatformToMove]->left =
          m_PlatformPositions[m_MoveRelativeToPlatform]-
>left +
          m_PlatformPositions[m_MoveRelativeToPlatform]-
>width +
          getRandomNumber(0, 20);
      }
      m_PlatformPositions[m_NextPlatformToMove]->width =
        getRandomNumber(20, 200);
      m_PlatformPositions[m_NextPlatformToMove]->height =
        getRandomNumber(10, 20);
      // Base the time to create the next platform
// on the width of the one just created
      m_PlatformCreationInterval =
        m_PlatformPositions[m_NextPlatformToMove]->width
 / 90;
      m_MoveRelativeToPlatform = m_NextPlatformToMove;
      m_NextPlatformToMove++;
      if (m_NextPlatformToMove == m_NumberOfPlatforms)
      {
        m_NextPlatformToMove = 0;
      }
      m_TimeSinceLastPlatform = 0;
    }
    // Has the player lagged behind the furthest back platform
bool laggingBehind = true;
    for (auto platformPosition : m_PlatformPositions)
    {
      if (platformPosition->left < m_PlayerPosition->left)
      {
        laggingBehind = false;
        break;// At least one platform is behind the player
      }
      else
      {
        laggingBehind = true;
      }
    }
    if (laggingBehind)
    {
```

```
        m_IsPaused = true;
        m_GameOver = true;
        SoundEngine::pauseMusic();
      }
    }
  }
```

We will break this lengthy code into a few sections, but please be sure to study it in its entirety, especially observing its structure of `if` statements and loops, to make the following discussion easier to understand. The `update` function receives the time that the previous iteration of the main game loop took to execute in the `timeSinceLastUpdate` variable.

The first part of the `update` function sets the following structure, which only runs when the game is not paused. Everything else we discuss happens inside this `if` statement, meaning that nothing happens when the game is paused:

```
  if (!m_IsPaused)
    {
```

Next in the `update` function is the following code, which only runs when the game is over – in other words, when the player has either just run the app or has just died and not restarted yet:

```
  if (m_GameOver)
  {
    m_GameOver = false;
    *m_CameraTime = 0;
    m_TimeSinceLastPlatform = 0;
    positionLevelAtStart();
  }
```

The preceding code sets `m_GameOver` to `false`, resets the timer, resets the time since the previous platform was spawned, and calls the `positionLevelAtStart` function, which we have already discussed. The net effect of this is that this block of code will only run once, and it does everything necessary to set a new game running (once the rest of the code is done).

Next in the `update` function, there is an `if` statement, as follows:

```
*m_CameraTime += timeSinceLastUpdate;
m_TimeSinceLastPlatform += timeSinceLastUpdate;
if (m_TimeSinceLastPlatform > m_PlatformCreationInterval)
  {
m_PlatformPositions[m_NextPlatformToMove]->top =
  m_PlatformPositions[m_MoveRelativeToPlatform]->top
+ getRandomNumber(-40, 40);
    …
```

In the preceding code, the `m_CameraTime` variable is incremented by the time that has passed since the last time `update` executed. This is the time that will eventually be displayed to the player. The `m_TimeSinceLastPlatform` is also incremented in the same way.

Next, there is an `if` statement that executes when `m_TimeSinceLastPlatform` exceeds `m_PlatformCreationInterval`. In other words, it's time to move a platform from behind the player to in front of the player. Then, the platform farthest behind the player (`m_NextPlatformToMove`) is randomly positioned relative to the platform furthest ahead of the player (`m_MoveRelativeToPlatform`) but only the height is adjusted at this point.

Also, within the preceding `if` statement, there is an `if-else` structure that will take care of horizontal positioning. Let's look at it again:

```
// How far away to create the next platform
// Bigger gap if lower than previous
if (m_PlatformPositions[m_MoveRelativeToPlatform]->top
< m_PlatformPositions[m_NextPlatformToMove]->top)
{
m_PlatformPositions[m_NextPlatformToMove]->left =
    m_PlatformPositions[m_MoveRelativeToPlatform]->left
+ m_PlatformPositions[m_MoveRelativeToPlatform]
->width + getRandomNumber(20, 40);
}
else
{
m_PlatformPositions[m_NextPlatformToMove]->left =
    m_PlatformPositions[m_MoveRelativeToPlatform]
->left +
    m_PlatformPositions[m_MoveRelativeToPlatform]
->width + getRandomNumber(0, 20);
}
```

In the preceding `if-else` statements, the `if` part checks how far away vertically the previous line of code has positioned the next platform. If it is lower, then the `if`-related code executes, and if it is higher, then the `else`-related code executes. The `else`-related code uses lower values for spacing the platforms horizontally, which makes sense because, if the platform is above the platform the player is on, the jumpable distance will be smaller.

Next, the following code executes:

```
m_PlatformPositions[m_NextPlatformToMove]->width =
getRandomNumber(20, 200);
m_PlatformPositions[m_NextPlatformToMove]->height =
getRandomNumber(10, 20);
// Base the time to create the next platform
// on the width of the one just created
m_PlatformCreationInterval =
    m_PlatformPositions[m_NextPlatformToMove]->width
/ 90;
```

```
m_MoveRelativeToPlatform = m_NextPlatformToMove;
m_NextPlatformToMove++;
if (m_NextPlatformToMove == m_NumberOfPlatforms)
{
m_NextPlatformToMove = 0;
}
m_TimeSinceLastPlatform = 0;
```

In the preceding code, a random width for the new platform is chosen, then a random height is chosen, and then an amount of time based on the randomly chosen width is initialized to `m_PlatformCreationInterval`. The next line increments the position in the vector for the next platform to be moved and the `if` statement that follows checks whether that value is beyond the last position in the vector and changes the value to zero (the first entry in the vector) if it is.

The final line of code above sets `m_TimeSinceLastPlatform` to zero so we can keep adding the time the loop took for each iteration until we eventually get to move another platform and do it all again.

At this point, we close the curly bracket of the `if (m_TimeSinceLastPlatform > m_PlatformCreationInterval)` block.

Following on and completing the `if(!m_Paused)` code and the entire `update` function, the following code checks to see if the disappearing platforms have caught up with the player (and therefore that they have failed the game):

```
// Has the player lagged behind the furthest back platform
bool laggingBehind = true;
for (auto platformPosition : m_PlatformPositions)
{
if (platformPosition->left < m_PlayerPosition->left)
{
```

```
    laggingBehind = false;
        break;// At least one platform is behind the player
    }
      else
      {
        laggingBehind = true;
      }
    }
    if (laggingBehind)
    {
      m_IsPaused = true;
      m_GameOver = true;
      SoundEngine::pauseMusic();
    }
```

In the preceding code, a `laggingBehind` Boolean is set to `true`. Next, the `for` loop goes through each of the platform positions checking if any of the platform's left-hand coordinates is less than (and, therefore, behind) the player. If any of them are, then the player still has a chance, and the `laggingBehind` Boolean is set to false.

If the `laggingBehind` variable remains set to `true`, then it means that all the platforms are in front, and the game is over. If `laggingBehind` is set to true, then the game is paused, the `m_GameOver` variable is set to `true`, and the music is paused.

Soon, we will code a menu that will allow the player to restart the game after they lose.

Finally, in the `update` function, we close the remaining curly braces of the decision structure (not shown again).

We are done explaining and coding the `update` function, but we are not quite ready to run our code because we have errors that refer to the

`PlayerUpdate` class. Furthermore, we haven't built any instances of the `LevelUpdate` class.

Next, we will code the basics of a player character by deriving an object from `Update` called `PlayerUpdate` and an object from `Graphics` called `PlayerGraphics`. Then, to finish the code for this chapter, we will add code to the factory that assembles all these different components and places them into `GameObject` instances that we can loop through each frame of the game. Furthermore, we will get to use the `InputReceiver` class with the `PlayerwUpdate` class and see how the responsibility for controlling the player is handled by the player-related classes.

Coding the player: Part 1

In this section, we will begin to create the controllable player character. We will make the character visible on the screen but will return to the `PlayerUpdate` and `PlayerGraphics` classes and add keyboard controls and animations.

Create two new classes: `PlayerUpdate`, which uses `Update` as the base class, and `PlayerGraphics`, which uses `Graphics` as the base class. After the next two sections, we will have a visible but not fully functioning player character.

Coding the PlayerUpdate class

Let's start with the `PlayerUpdate` class definition. Add the following code to `PlayerUpdate.h`:

```
#pragma once
#include "Update.h"
#include "InputReceiver.h"
#include <SFML/Graphics.hpp>
using namespace sf;
class PlayerUpdate :
    public Update
{
private:
    const float PLAYER_WIDTH = 20.f;
    const float PLAYER_HEIGHT = 16.f;
    FloatRect m_Position;
    bool* m_IsPaused = nullptr;
    float m_Gravity = 165;
    float m_RunSpeed = 150;
    float m_BoostSpeed = 250;
    InputReceiver m_InputReceiver;
```

```
    Clock m_JumpClock;
    bool m_SpaceHeldDown = false;
    float m_JumpDuration = .50;
    float m_JumpSpeed = 400;
public:
    bool m_RightIsHeldDown = false;
    bool m_LeftIsHeldDown = false;
    bool m_BoostIsHeldDown = false;
    bool m_IsGrounded;
    bool m_InJump = false;
    FloatRect* getPositionPointer();
    bool* getGroundedPointer();
    void handleInput();
    InputReceiver* getInputReceiver();
    // From Update : Component
void assemble(
        shared_ptr<LevelUpdate> levelUpdate,
        shared_ptr<PlayerUpdate> playerUpdate)
 override;
    void update(float fps) override;
};
```

In the preceding code, we have many variables and five functions. Let's run through them all:

- The constant float called `PLAYER_WIDTH` defines how wide the player will be in game units, and the constant float variable `PLAYER_HEIGHT` defines how high the player will be in game units.
- The `FloatRect` instance called `m_Position` will hold the player's location. We will soon see that this instance will be shared with any game entity that wants it. Platforms will use it for collision, and we saw in the previous section that the `LevelUpdate` class uses it for determining if the player has lagged behind the platforms to an extent that means the game is over.

- The Boolean pointer `m_IsPaused` will be used to connect to the `LevelUpdate` class variable, which pauses the game.
- The `float` variable `m_Gravity` is a value for pushing the player downward when not standing on a platform and moderating the upward force of boosting. The float variable `m_RunSpeed` is the speed at which the player can move left or right when grounded on a platform. The float `m_BoostSpeed` is the rate at which the player can move upward while boosting.
- The `InputReceiver` instance called `m_InputReceiver` is an instance of the `InputReceiver` class. Soon we will see how the `Factory` class connects `m_InputReceiver` to the `InputDispatcher` and, therefore, enables the `PlayerUpdate` class to access all the events of the keyboard and mouse.
- The `Clock` instance `m_JumpClock`, the Boolean `m_SpaceHeldDown`, the float variable `m_JumpDuration`, and the float variable `m_JumpSpeed` are all values that we will use to moderate how far and for how long the player jumps.
- The first variable in the private section is the Boolean `m_RightIsHeldDown`, followed by more Booleans, `m_LeftIsHeldDown`, `m_BoostIsHeldDown`, `m_IsGrounded`, and `m_InJump`. All these values can be set and unset according to how the player interacts with the keyboard. They can then be responded to in the `update` function.
- The `getPositionPointer` function returns a `FloatRect` pointer that provides access to `m_Position` for any other class that wants it. This function will be called in the factory by the classes that need to.
- The `getGroundedPointer` function returns a pointer to a Boolean that shares whether the player is currently grounded as determined by the `m_IsGrounded` Boolean variable.

- The `handleInput` function will use the `InputReceiver` instance to handle all the input data received in each frame from the `InputDispatcher` instance in the main game loop.
- The `getInputReceiver` function returns a pointer to an `InputReceiver` instance. Just one line of code will be required to implement this, but it will crucially allow the `InputDispatcher` instance in the `main` function to share all the events with the `PlayerUpdate` class.
- The `assemble` function is our implementation of the pure virtual function from the `Update` class. The parameters are `shared_ptr<LevelUpdate> levelUpdate` and `shared_ptr<PlayerUpdate> playerUpdate`. This means we can prepare the `PlayerUpdate` class for action by calling any public functions of the `LevelUpdate` class. Of course, passing `PlayerUpdate` to itself is unnecessary but is a symptom of implementing the simplest possible version of the Entity Component system.
- The `update` function is our implementation of the pure virtual function from the `Update` class and simply receives the time the game loop took to execute. What we do with this time in the function implementation will be more interesting.

As we have quite a lot of code to add to `PlayerUpdate.cpp`, we will do it in three steps. First, add the following code to `PlayerUpdate.cpp`:

```
#include "PlayerUpdate.h"
#include "SoundEngine.h"
#include "LevelUpdate.h"
FloatRect* PlayerUpdate::getPositionPointer()
{
    return &m_Position;
}
bool* PlayerUpdate::getGroundedPointer()
```

```
{
    return &m_IsGrounded;
}
InputReceiver* PlayerUpdate::getInputReceiver()
{
    return &m_InputReceiver;
}
```

In the first part of the `PlayerUpdate` code, we add the required include directives. The `getPositionPointer` function returns the address of the `FloatRect` instance that holds the position of the player. The `getGroundedPointer` function returns the address of the Boolean that detects whether the player is currently standing on a platform. Interestingly, the platforms will determine and set the value of this Boolean (using this pointer) and the `PlayerGraphics` class will use the value to make decisions about animations (using this pointer). The `getInputReceiver` function returns a pointer to the `InputReceiver` instance allowing the `InputDispatcher` to connect to and send all the required event data.

For step 2, add the following code:

```
void PlayerUpdate::assemble(
    shared_ptr<LevelUpdate> levelUpdate,
    shared_ptr<PlayerUpdate> playerUpdate)
{
    SoundEngine::SoundEngine();
    m_Position.width = PLAYER_WIDTH;
    m_Position.height = PLAYER_HEIGHT;
    m_IsPaused = levelUpdate->getIsPausedPointer();
}
```

In the second part of the `PlayerUpdate` code, we coded the `assemble` function. Notice, as mentioned previously, that the `PlayerUpdate` parameter

is unused. The `SoundEngine` class is initialized and ready to play some sounds, the position and height are initialized, and, most interestingly, the `LevelUpdate` shared pointer is used to call the `getIsPausedPointer` function and initialize `m_IsPaused`. Now the `PlayerUdate` class can always check if the game is ever paused.

Next, you will need to add the third and final piece of code (in this chapter) for `PlayerUpdate.cpp`:

```
void PlayerUpdate::handleInput()
{
    m_InputReceiver.clearEvents();
}
void PlayerUpdate::update(float timeTakenThisFrame)
{
    handleInput();
}
```

In the third and final part of the `PlayerUpdate.cpp` code that we added, the `handleInput` function calls the `clearEvents` function on the `m_InputReceiver` instance. This clears the events ready for the next iteration of the loop. This doesn't achieve anything because we haven't read any events yet, but we will get to that in *Chapter 18*. Finally, we added the `update` function. All it does is call the function we just coded; however, in *Chapter 18*, we will code a fully functioning and responsive player character.

Coding the PlayerGraphics class

We have put in the bare bones of behavior for our player character. Next, we will begin to code the appearance by extending the `Graphics` class with

a `PlayerGraphics` class. As with the `PlayerUpdate` class, we will just start with the basics and build on it as the project progresses. Add the following code to `PlayerGraphics.h`:

```
#pragma once
#include "Graphics.h"
// We will come back to this soon
//class Animator;
class PlayerUpdate;
class PlayerGraphics : public Graphics
{
private:
    FloatRect* m_Position = nullptr;
    int m_VertexStartIndex = -999;
    // We will come back to this soon
//Animator* m_Animator;
    IntRect* m_SectionToDraw = new IntRect;
    IntRect* m_StandingStillSectionToDraw = new IntRect;
    std::shared_ptr<PlayerUpdate> m_PlayerUpdate;
    const int BOOST_TEX_LEFT = 536;
    const int BOOST_TEX_TOP = 0;
    const int BOOST_TEX_WIDTH = 69;
    const int BOOST_TEX_HEIGHT = 100;
    bool m_LastFacingRight = true;
public:

    //From Component : Graphics
void assemble(VertexArray& canvas,
        shared_ptr<Update> genericUpdate,
        IntRect texCoords) override;
    void draw(VertexArray& canvas) override;
};
```

Notice a couple of commented-out references to an `Animator` class. Later, in *Chapter 18*, we will animate the player to make it look like it is running. We will also animate the flames on the fireballs. To be able to run the code

without errors as soon as possible, the preceding code has the `Animator` class commented out.

In the preceding code, we have the following variables and function declarations. Let's go through them one at a time:

- The `FloatRect` variable `m_Position` is set to `nullptr`. This will represent the position of the player. The int variable `m_VertexStartIndex` will hold the position in the `VertexArray` where the quad representing the player will start. So, when it comes to moving the player, we will know that the vertices we are interested in will be `m_VertexStartIndex`, `m_VertexStartIndex+1`, `m_VertexStartIndex+2`, and `m_VertexStartIndex+3`.
- The `m_SectionToDraw` variable is an `IntRect` pointer. It will hold the integer texture coordinates within the texture atlas of the current frame of animation for the player. The `Animate` class will manipulate these values as required. We will code the `Animate` class in *Chapter 18*.
- The `IntRect` pointer `m_StandingStillSectionToDraw` will hold the texture coordinates for when the player is not running.
- The `shared_ptr<PlayerUpdate> m_PlayerUpdate` variable is a pointer to the `PlayerUpdate` instance. Holding a pointer to the `PlayerUpdate` instance will allow this class to call all the public functions and read all the public variables of the `PlayerUpdate` class.
- The `const int BOOST_TEX_LEFT`, `BOOST_TEX_TOP`, `BOOST_TEX_WIDTH`, and `BOOST_TEX_HEIGHT` are initialized with the coordinates of the frame of animation representing the player boosting, within the texture atlas.
- The Boolean `m_LastFacingRight` is initialized to true and will keep track of the player switching the direction they face. This will be needed when animating.

- The `assemble` function is the overridden implementation from the `Graphics` class. The `assemble` function receives a `VertexArray` reference called `canvas`, and a `shared_ptr<Update>` called `genericUpdate`. It will be interesting to see what we do with `genericUpdate`. In each different `Update` derived class, we will see how we convert it to the specific `Update` variant required and, therefore, provide access to its public functions. The `assemble` function also receives an `IntRect` instance called `texCoords` that will hold the texture coordinates for the graphic in the texture atlas.
- The `draw` function receives the `VertexArray` as a reference and is called each frame. This enables the `draw` function to handle moving vertex or texture coordinates as required in each frame of the game.

Let's code all these functions and begin to use the variables we have been discussing. Add the following code to `PlayerGraphics.cpp`:

```
#include "PlayerGraphics.h"
#include "PlayerUpdate.h"
void PlayerGraphics::assemble(
    VertexArray& canvas,
    shared_ptr<Update> genericUpdate,
    IntRect texCoords)
{
    m_PlayerUpdate =
        static_pointer_cast<PlayerUpdate>(genericUpdate);
    m_Position =
        m_PlayerUpdate->getPositionPointer();
    m_VertexStartIndex = canvas.getVertexCount();
    canvas.resize(canvas.getVertexCount() + 4);
    canvas[m_VertexStartIndex].texCoords.x =
        texCoords.left;
    canvas[m_VertexStartIndex].texCoords.y =
        texCoords.top;
    canvas[m_VertexStartIndex + 1].texCoords.x =
        texCoords.left + texCoords.width;
```

```
        canvas[m_VertexStartIndex + 1].texCoords.y =
            texCoords.top;
        canvas[m_VertexStartIndex + 2].texCoords.x =
            texCoords.left + texCoords.width;
        canvas[m_VertexStartIndex + 2].texCoords.y =
            texCoords.top + texCoords.height;
        canvas[m_VertexStartIndex + 3].texCoords.x =
            texCoords.left;
        canvas[m_VertexStartIndex + 3].texCoords.y =
            texCoords.top + texCoords.height;
    }
    void PlayerGraphics::draw(VertexArray& canvas)
    {
        const Vector2f& position =
            m_Position->getPosition();
        const Vector2f& scale =
            m_Position->getSize();
        canvas[m_VertexStartIndex].position =
            position;
        canvas[m_VertexStartIndex + 1].position =
            position + Vector2f(scale.x, 0);
        canvas[m_VertexStartIndex + 2].position =
            position + scale;
        canvas[m_VertexStartIndex + 3].position =
            position + Vector2f(0, scale.y);
    }
```

Some of the preceding code might look familiar because we are assigning vertex and texture coordinates to an SFML `VertexArray` instance, just like we did for the background in the zombie game. We will be doing this or something similar in every `Graphics` derived class.

Here, though, we are working in a different context to the zombie game, so let's run through how all the code we have just added works by splitting it up into four parts.

First, we have this function signature:

```
void PlayerGraphics::assemble(
   VertexArray& canvas,
   shared_ptr<Update> genericUpdate,
   IntRect texCoords)
{
  …
…
}
```

The first part of the `PlayerGraphics.cpp` we are looking at is the signature for the `assemble` function. As a reminder, the `assemble` function as already discussed is the overridden implementation from the `Graphics` class. The `assemble` function receives a `VertexArray` reference called `canvas` and `a shared_ptr<Update>` called `genericUpdate`. The `assemble` function also receives an `IntRect` instance called `texCoords` that will hold the texture coordinates for the graphic in the texture atlas.

Second, inside the curly braces of the `assemble` function, we have this:

```
m_PlayerUpdate =
static_pointer_cast<PlayerUpdate>(genericUpdate);
m_Position =
m_PlayerUpdate->getPositionPointer();
m_VertexStartIndex = canvas.getVertexCount();
canvas.resize(canvas.getVertexCount() + 4);
canvas[m_VertexStartIndex].texCoords.x =
texCoords.left;
canvas[m_VertexStartIndex].texCoords.y =
texCoords.top;
canvas[m_VertexStartIndex + 1].texCoords.x =
texCoords.left + texCoords.width;
canvas[m_VertexStartIndex + 1].texCoords.y =
texCoords.top;
canvas[m_VertexStartIndex + 2].texCoords.x =
texCoords.left + texCoords.width;
canvas[m_VertexStartIndex + 2].texCoords.y =
texCoords.top + texCoords.height;
```

```
canvas[m_VertexStartIndex + 3].texCoords.x =
texCoords.left;
canvas[m_VertexStartIndex + 3].texCoords.y =
texCoords.top + texCoords.height;
```

The second part of the `PlayerGraphics.cpp` file (above) in the code for the `assemble` function uses the `static_pointer_cast` function to turn the base class `Update` instance into a child class `PlayerUpdate` instance and then saves the result into `m_PlayerUpdate`.

Next, we initialize `m_VertexStartIndex` by calling `canvas.getVertexCount`, then add enough space for another quad to the vertex array by calling `canvas.resize`. Following on, in the next eight lines of code, we initialize all the player character's texture coordinates in the `VertexArray` using the `IntRect texCoords` that was passed in as a parameter.

Third and finally, we have this code:

```
void PlayerGraphics::draw(VertexArray& canvas)
{
const Vector2f& position =
        m_Position->getPosition();
const Vector2f& scale =
        m_Position->getSize();
canvas[m_VertexStartIndex].position =
        position;
canvas[m_VertexStartIndex + 1].position =
        position + Vector2f(scale.x, 0);
canvas[m_VertexStartIndex + 2].position =
        position + scale;
canvas[m_VertexStartIndex + 3].position =
        position + Vector2f(0, scale.y);
}
```

The third and final part of the `PlayerGraphics.cpp` file is the `draw` function. For now, we just use the `m_Position.getPosition` function to initialize a `Vector2f` instance called `position` and the `getSize` function to initialize `Vector2f` called `scale`. We then use `position` and `scale` to set the positions of the player character's vertices in the `VertexArray`.

We will complete the updating and input handling of the `PlayerUpdate` class as well as the animation and the `Animator` class for the `PlayerGraphics` class once we have added some camera-related classes to see the player properly as well as platforms for the player to run upon.

Coding the factory to use all our new classes

The `Factory` is an important class. It will be where we create all our smart pointers to derived `Update` and `Graphics` instances. We will call all the constructors and `assemble` function implementations while sharing the various required pointers that we have been coding. For example, the `Factory` is where we will share the pointer to the player and the position of the platforms with the `LevelUpdate` instance.

Remembering the texture coordinates

First of all, we will add code to the `Factory.h` file. In the `Factory.h` file, add the following variables to the `private` section:

```
const int PLAYER_TEX_LEFT = 0;
 const int PLAYER_TEX_TOP = 0;
 const int PLAYER_TEX_WIDTH = 80;
 const int PLAYER_TEX_HEIGHT = 96;
```

```
    const float CAM_VIEW_WIDTH = 300.f;
    const float CAM_SCREEN_RATIO_LEFT = 0.f;
    const float CAM_SCREEN_RATIO_TOP = 0.f;
    const float CAM_SCREEN_RATIO_WIDTH = 1.f;
    const float CAM_SCREEN_RATIO_HEIGHT = 1.f;
    const int CAM_TEX_LEFT = 610;
    const int CAM_TEX_TOP = 36;
    const int CAM_TEX_WIDTH = 40;
    const int CAM_TEX_HEIGHT = 30;
    const float MAP_CAM_SCREEN_RATIO_LEFT = 0.3f;
    const float MAP_CAM_SCREEN_RATIO_TOP = 0.84f;
    const float MAP_CAM_SCREEN_RATIO_WIDTH = 0.4f;
    const float MAP_CAM_SCREEN_RATIO_HEIGHT = 0.15f;
    const float MAP_CAM_VIEW_WIDTH = 800.f;
    const float MAP_CAM_VIEW_HEIGHT = MAP_CAM_VIEW_WIDTH / 2;
    const int MAP_CAM_TEX_LEFT = 665;
    const int MAP_CAM_TEX_TOP = 0;
    const int MAP_CAM_TEX_WIDTH = 100;
    const int MAP_CAM_TEX_HEIGHT = 70;
    const int PLATFORM_TEX_LEFT = 607;
    const int PLATFORM_TEX_TOP = 0;
    const int PLATFORM_TEX_WIDTH = 10;
    const int PLATFORM_TEX_HEIGHT = 10;
    const int TOP_MENU_TEX_LEFT = 770;
    const int TOP_MENU_TEX_TOP = 0;
    const int TOP_MENU_TEX_WIDTH = 100;
    const int TOP_MENU_TEX_HEIGHT = 100;
    const int RAIN_TEX_LEFT = 0;
    const int RAIN_TEX_TOP = 100;
    const int RAIN_TEX_WIDTH = 100;
    const int RAIN_TEX_HEIGHT = 100;
```

We will use all the new constant values as we proceed through the rest of the project. The variables represent the texture coordinates of all the images in the texture atlas. Often, you would load these values from a file, but this suits our purposes fine.

Finally, to use these new classes, we will instantiate and configure them in our `Factory`.

Add these additional highlighted directives to the `Factory.cpp`, so we can use our newly created classes:

```
#include "Factory.h"
#include "LevelUpdate.h"
#include "PlayerGraphics.h"
#include "PlayerUpdate.h"
#include "InputDispatcher.h"
```

Add this code to the `loadLevel` function to instantiate a `LevelUpdate` instance inside a `GameObject` instance into the `Factory.cpp` file:

```
// Build a level game object
GameObject level;
shared_ptr<LevelUpdate> levelUpdate =
  make_shared<LevelUpdate>();
level.addComponent(levelUpdate);
gameObjects.push_back(level);
```

There is quite a lot to discuss in the preceding snippet of code but, as we will see, it is not as complicated as it might first appear.

The code creates a new `GameObject` instance called `level`. Next, we create a shared pointer of the `LevelUpdate` type. Following on, we call the `addComponent` function on `level` and pass in the `LevelUpdate` instance, level. Finally, we call the `push_back` function on our `gameObjects` vector. This is a significant step because it means we finally have a functioning `GameObject` instance that will be looped over for each frame of the game loop. Astute readers might have noticed that we didn't call the `assemble` function yet. We will get to that soon.

Next, add this code to instantiate a `PlayerGraphics` and a `PlayerUpdate` instance inside a `GameObject` instance to the `loadLevel` function:

```
// Build a player object
GameObject player;
shared_ptr<PlayerUpdate> playerUpdate =
  make_shared<PlayerUpdate>();
playerUpdate->assemble(levelUpdate, nullptr);
player.addComponent(playerUpdate);
inputDispatcher.registerNewInputReceiver(
  playerUpdate->getInputReceiver());
shared_ptr<PlayerGraphics> playerGraphics =
  make_shared<PlayerGraphics>();
playerGraphics->assemble(canvas, playerUpdate,
  IntRect(PLAYER_TEX_LEFT, PLAYER_TEX_TOP,
    PLAYER_TEX_WIDTH, PLAYER_TEX_HEIGHT));
player.addComponent(playerGraphics);
gameObjects.push_back(player);
// Make the LevelUpdate aware of the player
levelUpdate->assemble(nullptr, playerUpdate);
```

In the preceding code, we create another `GameObject` instance called `player` and a `PlayerUpdate` shared pointer called `playerUpdate`. We call the `assemble` function on `playerUpdate` and pass in the required parameters. These required parameters are the `LevelUpdate` shared pointer, but we pass `nullptr` where we should pass a `PlayerUpdate` pointer. This is, as mentioned previously, because of the simplification of the Entity Component system we are using. The `PlayerUpdate` class obviously does not need a copy of itself.

Then, we call the `addComponent` function on `player` and call `registerNewInputReceiver` on the `InputDispatcher` instance. Notice that we pass in the required value by calling `getInputReceiver` on the `PlayerUpdate` instance. At this point, not only do we have another

`GameObject` instance almost ready to be iterated in the game loop in the `gameObjects` vector but we have also established a connection to all the operating system events provided by SFML. Now we can move on to the `PlayerGraphics` class instance.

Next, we instantiate an instance of `PlayerGraphics` and call the `assemble` function, passing in the `VertexArray`, the `LevelUpdate` instance, and the texture coordinates. Now we add the `GraphicsComponent` instance into the `GameObject` instance with the `addComponent` function and call `push_back` to add the player character's `GameObject` into the `gameObjects` vector.

The final line of code calls the `assemble` function on `levelUpdate` because we couldn't do so previously as the `PlayerUpdate` instance didn't exist yet. This is the kind of knowledge that the `Factory` class is expected to have.

Running the game

If we run the game at this point, we still get the blank gray screen. This is because we are not drawing our `VertexArray`. In the next chapter, we will see how to draw the `VertexArray` twice to create a regular view as well as a mini map. We will achieve this by coding some classes to represent cameras or views of our game. For now, just add this highlighted line of code to the `main` function in `Run.cpp` just before the call to `window.display`:

```
// Temporary code until next chapter
window.draw(canvas, factory.m_Texture);
// Show the new frame.
window.display();
```

Now, if you run the game and look closely, very closely, in the top-left corner of the screen, you can just about see a tiny, static player graphic. I

haven't provided a screenshot because it is so tiny. Read on for a solution. You will also have noticed the short piece of music playing on a loop. If you prefer to work in silence while you test your code going forward, just delete these two lines of code from the `LevelUpdate assemble` function as they were just there for testing purposes anyway:

```
 //temp
  SoundEngine::startMusic();
```

We will handle starting and stopping the music properly when we code a menu for our game.

If you want to get a closer look at the player graphic, temporarily edit the `assemble` function in the `PlayerUpdate.cpp` file to increase the size of the player, as shown with these two highlighted lines of code:

```
void PlayerUpdate::assemble(
    shared_ptr<LevelUpdate> levelUpdate,
    shared_ptr<PlayerUpdate> playerUpdate)
{
    SoundEngine::SoundEngine();

    m_Position.width = PLAYER_WIDTH * 10;
    m_Position.height = PLAYER_HEIGHT * 10;
    m_IsPaused = levelUpdate->getIsPausedPointer();
}
```

Run the game and you can clearly see the player in the top-left corner of the screen, as shown next:

Figure 16.1: Enlarged player image

Be sure to remove the `* 10` from the two previously edited lines of code.

We could have fairly easily added some code to make the player character controllable, but the chapter was getting a bit long, and we will do so in a couple of chapters' time when we have our camera game objects working and we can see the player better.

Summary

In this chapter, we have achieved a lot. We have coded a sound-related class that also plays music in a loop, and we have coded a class that handles all the game logic and encapsulated it in a `GameObject` that runs once per loop of the game loop.

We have coded the beginning of a playable character using a graphics component and an update component composed within a game object. This is the essence of the Entity Component system. This process will be repeated for every type of entity in our game.

We have continued coding the factory, which is responsible for assembling all the different game objects and sharing the appropriate data between them.

In the next chapter, we will focus on graphics and drawing by coding `CameraGraphics` and `CameraUpdate` classes that are also derived from `Graphics` and `Update`.

17

Graphics, Cameras, Action

We need to talk in depth about the way the graphics will work in this project. As we will be coding the cameras that do the drawing in this chapter, now seems like a good time to talk about the graphics too. If you look in the `graphics` folder, there is just one graphic. Furthermore, we are not calling `window.draw` at any point in our code so far. We will discuss why draw calls should be kept to a minimum, as well as implement our `Camera` classes that will handle this for us. Finally, by the end of this chapter, we will be able to run the game and see the cameras in action, including the main view, the radar view, and the timer text.

The completed code for this chapter is in the `Run3` folder.

Here is what is coming up in this chapter:

- Cameras, draw calls, and SFML View
- Coding the `camera` classes
- Adding camera instances to the game
- Running the game

The code for this chapter is in the `Run3` folder.

Cameras, draw calls, and SFML View

In all our previous projects, all the entities in our games (with one exception) were graphically represented with a **sprite**. This is fine when there are only a few, a few dozen, or even a few hundred entities being drawn. It is important because the speed that SFML can draw each frame of our game has a direct relationship to the number of times we call `window.draw`. This is not an SFML flaw but is directly connected to how OpenGL uses the graphics card.

The reason is that each time we call draw, quite a lot happens behind the scenes to set up OpenGL so that it is ready to draw. To quote the SFML website,

> *"...each call [to draw] involves setting a set of OpenGL states, resetting matrices, changing textures, etc. All of this is required even when simply drawing two triangles (a sprite)."*

So, using a vertex array and rendering multiple images with one draw call to do our drawing, whenever possible, is a very good idea. This means we need to change how we deal with graphics in general. Instead of having a sprite for every game object, we will now have a starting index in a vertex array, and instead of hundreds of sprites and many textures, we will have a single vertex array and one texture containing all the graphics. Furthermore, consider that every time we have drawn the score, the time, or any other text to the screen, we have also made a draw call. The SFML text-related classes are so useful that we will not try and stop using them, and we only have one

place in this project where we will use SFML Text anyway. I am referring to the time in the top-left corner of the screen.

The menu-related text is static and does not need to be computed so it is drawn from images within the texture atlas. If you are wondering how you would keep your draw calls to a minimum if you had lots of text to display, the answer is that you would treat your text in the same way you do regular graphics and provide a texture atlas of the alphabet, numbers 0 through 9, and any punctuation you require. This is way more complex than using the SFML `Text` class but not too complex.

All it requires is that each character (number, letter, etc.) being drawn in the current frame has an index in the vertex array (and related coordinates) and you would then parse the strings you want to draw to the screen and layer the appropriate vertex positions with the appropriate texture coordinates. By the time you complete this project, you will have done this half a dozen times or more and it won't be a mystery. Of course, we used a vertex array for the background of the zombie game and we have already used texture coordinates combined with a vertex array index for the player graphic in the preceding chapter.

We will now code the cameras. The cameras will handle the calling of the `draw` function whenever required. There will be two cameras. The regular camera will call `draw` twice: once for the vertex array and once for the timer in the top-left corner. The mini-map camera will call `draw` once for a differently oriented view of the world showing a radar-like view to the player.

To achieve this, each camera will need access to the same `RenderWindow` instance and will need to adjust the settings to the SFML `View` instance,

which defines the camera's position on screen, the length-to-width ratio, and the zoom level for the intended purpose.

Coding the camera classes

We will have two cameras in our game, and each will have an `Update` derived class and a `Graphics` derived class. The `CameraUpdate` class will handle movement to follow the player and interaction with the operating system via an `InputReceiver` instance. The `CameraGraphics` class will handle all the drawing by referring to the data in `CameraUpdate` and holding a copy of the texture atlas, the RenderWindow instance, and an SFML `Text` object. Later, in *Chapter 21*, we will introduce a couple more features (and `draw` calls) to add a parallax background and a neat shader effect.

Coding the CameraUpdate class

Create two new classes to represent the cameras: `CameraUpdate`, which has a base class of `Update`, and `CameraGraphics`, which has a base class of `Graphics`. As we are coming to expect, these classes will be wrapped/composed in a `GameObject` instance for use in our game loop.

Add the following code to the `CameraUpdate.h` file:

```
#pragma once
#include "Update.h"
#include "InputReceiver.h"
#include <SFML/Graphics.hpp>
using namespace sf;
class CameraUpdate :
    public Update
{
private:
    FloatRect m_Position;
```

```
    FloatRect* m_PlayerPosition;
    bool m_ReceivesInput = false;
    InputReceiver* m_InputReceiver = nullptr;
public:
    FloatRect* getPositionPointer();
    void handleInput();
    InputReceiver* getInputReceiver();
    //From Update : Component
void assemble(shared_ptr<LevelUpdate> levelUpdate,
        shared_ptr<PlayerUpdate> playerUpdate) override;
    void update(float fps) override;
};
```

In the preceding code, the `FloatRect` named `m_Position` is declared ready to hold the position of the camera. A floating-point rectangle is perfect for holding the top, left, length, and width of a camera view in world space as opposed to integer pixel positions.

The `FloatRect` named `m_PlayerPosition` is a pointer and will be initialized to the address of the `FloatRect` instance in the `PlayerUpdate` class. This will make it possible for the `CameraUpdate` class to follow the player around the world wherever it goes.

The `bool` variable `m_ReceivesInput` is useful because only one of our cameras will receive input. We don't need to bother with the extra overhead of receiving and handling input in the main camera, just in the mini-map camera. This is because the main camera just follows the player character around and is not controlled by the player. By default, this Boolean is initialized to `false`.

The `InputReceiver` pointer `m_InputReceiver` is for registering with the `InputDispatcher`. By default, it is set to `nullptr` because, as mentioned previously, only one of our two cameras will need it.

The `getPositionPointer` function will enable the `CameraGraphics` class to track the `CameraUpdate` class by returning the address of the `FloatRect` instance, which defines the view of the camera. So, in summary, this class (`CameraUpdate`) will track the player and the `CameraGraphics` class will track this class.

The `handleInput` function will be called once each iteration of the game loop from the `update` function but only in the camera that needs it.

The `getInputReceiver` function is called from the factory by the `InputDispatcher` class. It can then access and store a pointer to the `InputReceiver` in `CameraUpdate`.

The `assemble` function will prepare this class for action. As a reminder, the parameters are a `shared_ptr<LevelUpdate>` called `levelUpdate`, and a `shared_ptr<PlayerUpdate>` called `playerUpdate`. This is the first function we override from the `Update` class. Exactly how we use these parameters we will see very soon.

The `update` function is called once each frame and it receives the duration of the main game loop. We can now move on to see how we implement these functions and use all these variables.

Let's code the implementations of the functions in a couple of parts. To get started, add the following to the `CameraUpdate.cpp` file.

```
#include "CameraUpdate.h"
#include "PlayerUpdate.h"
FloatRect* CameraUpdate::getPositionPointer()
{
    return &m_Position;
}
void CameraUpdate::assemble(
    shared_ptr<LevelUpdate> levelUpdate,
```

```
    shared_ptr<PlayerUpdate> playerUpdate)
{
    m_PlayerPosition =
        playerUpdate->getPositionPointer();
}
InputReceiver* CameraUpdate::getInputReceiver()
{
    m_InputReceiver = new InputReceiver;
    m_ReceivesInput = true;
    return m_InputReceiver;
}
```

The `getPositionPointer` function simply returns the address of the `m_Position` variable. The `assemble` function simply stores the address of the player's position in `m_PlayerPosition` so it can always be referred to.

The `getInputReceiver` function initializes a new `InputReceiver` instance, sets the `m_ReceivesInput` variable to `true`, and then returns the address of the newly created instance. The effect of this function is that our code will only initialize and share an `InputReceiver` instance if we call it. Therefore, in the `Factory` class, we can easily pick and choose which cameras handle input and which don't. By setting the `m_ReceivesInput` variable to `true` and because the default value is `false`, each instance of the class will know whether it handles input or not. In short, each class will be told and prepared to handle input or not; the class itself doesn't have to choose.

For the next part of the `CameraUpdate` class, add the following to `CameraUpdate.cpp` right after the previous code.

```
void CameraUpdate::handleInput()
{
    m_Position.width = 1.0f;
    for (const Event& event : m_InputReceiver->getEvents())
    {
        // Handle mouse wheel event for zooming
```

```
if (event.type == sf::Event::MouseWheelScrolled)
        {
            if (event.mouseWheelScroll.wheel ==
                sf::Mouse::VerticalWheel)
            {
                // Accumulate the zoom factor based on delta
                m_Position.width *=
                    (event.mouseWheelScroll.delta > 0)
                    ? 0.95f : 1.05f;
            }
        }
        m_InputReceiver->clearEvents();
    }
}
```

In the preceding code, the `handleInput` function listens for the mouse wheel being scrolled. The `m_Position.width` value is set to `1` and we will see why in a moment. The code shown next loops through all the events, just as we have in every game so far, with the only difference being that we capture the events by calling the `m_InputReceiver->getEvents` function.

Inside the event loop, we care about just one event. That event is `sf::Event::MouseWheelScrolled`. When that event is detected, this `if` statement is executed:

```
if (event.mouseWheelScroll.wheel ==
                sf::Mouse::VerticalWheel)
```

The preceding statement checks if the mouse wheel was scrolled, and if it was, this next line of code is executed.

```
m_Position.width *=
(event.mouseWheelScroll.delta > 0)
   ? 0.95f : 1.05f;
```

This line of code modifies the `m_Position.width` value based on the direction of mouse wheel scrolling.

The value held in `event.mouseWheelScroll.delta` is a value that describes the amount by which the mouse wheel was scrolled. If the value is positive, it means the wheel was scrolled upward, and if it's negative, it means the wheel was scrolled downward.

The expression `(event.mouseWheelScroll.delta > 0)` is a ternary conditional operator. It checks if the value is greater than 0. If it is, the expression evaluates to `true`; otherwise, it evaluates `false`. Depending on the result of the operator, one of two values is chosen:

- If `delta > 0`, meaning the mouse wheel was scrolled upward, then `0.95f` is chosen.
- If `delta <= 0`, meaning the mouse wheel was scrolled downward, then `1.05f` is chosen.

The chosen value (`0.95f` or `1.05f`) is then multiplied by `m_Position.width`, which we previously initialized to `1`. If the result is greater than 0, it decreases `m_Position.width` by 5%, and if the result is less than 0, it increases `m_Position.width` by 5%. If the scroll wheel was not touched, the value remains at exactly `1`. We will see what we do with this value in the `update` function very soon. The final line of code in the `handleInput` function clears all the events ready for the next frame of the game.

Finally, for the `CameraUpdate` class, add the `update` function to `CameraUpdate.cpp`.

```
void CameraUpdate::update(float fps)
{
    if (m_ReceivesInput)
    {
```

```
            handleInput();
            m_Position.left = m_PlayerPosition->left;
            m_Position.top = m_PlayerPosition->top;
        }
        else
        {
            m_Position.left = m_PlayerPosition->left;
            m_Position.top = m_PlayerPosition->top;
            m_Position.width = 1;
        }
    }
```

The `update` function checks if this instance receives input. If it does, it calls `handleInput`. This means that any `CameraUpdate` instances that we call `getInputReceiver` on will execute this code. In the `if` block `m_Position`, `left` and `top` variable values are set to the same position as the player's. Note that the key part of the code is what is missing. We do not set the width. This means whatever value we set for `m_Position.width` back in the `handleInput` function remains. When the `CameraGraphics` class sets the parameters on the SFML `View` instance at each frame, this will have the effect of zooming in and out in sync with the mouse scroll wheel.

In the `else` block, we do the same as in the `if` block but additionally set `m_Position.width` to `1`. When the `else` block executes, the `CameraGraphics` class will not cause any zoom. Let's move on to the `CameraGraphics` class now.

Coding the CameraGraphics class part 1

We have seen how the `CameraUpdate` class works, how it conditionally responds to scrolling the mouse wheel, and that it stores that movement in its width. We have also seen how it stores the player's position in its `left` and `top` variables. Furthermore, we learned that this class (`CameraGraphics`) will use all these values. Let's see how all this comes together. Add the following to `CameraGraphics.h`.

```
#pragma once
#include "SFML/Graphics.hpp"
#include "Graphics.h"
using namespace sf;
class CameraGraphics :
public Graphics
{
private:
RenderWindow* m_Window;
View m_View;
int m_VertexStartIndex = -999;
Texture* m_Texture = nullptr;
FloatRect* m_Position = nullptr;
bool m_IsMiniMap = false;
// For zooming the mini map
const float MIN_WIDTH = 640.0f;
const float MAX_WIDTH = 2000.0f;
// For the Time UI
Text m_Text;
Font m_Font;
int m_TimeAtEndOfGame = 0;
float m_Time = 0;
public:
CameraGraphics(RenderWindow* window,
Texture* texture,
Vector2f viewSize,
FloatRect viewport);
float* getTimeConnection();
```

```
    // From Component : Graphics
    void assemble(VertexArray& canvas,
    shared_ptr<Update> genericUpdate,
    IntRect texCoords) override;
    void draw(VertexArray& canvas) override;
    };
```

That's quite a lot of code, so let's go through it. In the `private` section of the `CameraGraphics.h` file, we declare a pointer to a `RenderWindow` called `m_Window`, and a `View` instance called `m_View;` these are not normally present in our `Graphics` derived classes. The reason the `CameraGraphics` class needs them is it will be responsible for drawing the `VertexArray` that will contain the updated vertex positions and texture coordinates in each frame. This makes sense because the camera can control moving and zooming and then drawing. We can have as many cameras as we like. We could make a four-player game with the screen divided into four, a two-player split-screen game, or, as we are going to do, a full-screen camera and a mini-map/radar-like view.

The integer `m_VertexStartIndex` will hold the starting index within the vertex array of the quad for the camera. You might be thinking that the camera just draws the vertex array, so why should it need its own quad? You would be quite right to wonder why as often the camera will not have its own quad and texture coordinates, but our camera will have an almost fully transparent rectangle to create the border between the mini map and the main screen.

The `Texture` pointer `m_Texture` is the texture that holds our image with everything in it. The `m_Position` variable is a `FloatRect` that holds the size and coordinates of the camera's view of the world.

The Boolean `m_IsMiniMap` will help us write code that varies slightly between the main view and the mini-map view. You could easily build two separate classes, say `MainCameraGraphics` and `RadarCameraGraphics`, and avoid a few `if` statements in the code if you wish.

The constants `MIN_WIDTH` and `MAX_WIDTH` lock the minimum and maximum sizes for the view of the world, and this is necessary because we will be writing code that allows the mini map to be zoomed in and out.

The `Text`, `Font`, and `float m_Time` members are for displaying the time in the top-left corner of the screen. There will be three `draw` calls in each frame of the game, one for each camera and one for the text, but we will only call `draw` for the text in the main camera. Later, in *Chapter 21*, we will add a fourth draw call when we add a parallax background.

The integer `m_TimeAtEndOfGame` helps us show the time after the game has ended.

In the `public` section of the `CameraGraphics.h` file, we have the constructor function declaration. The constructor takes parameters for initializing the `RenderWindow` pointer and the `Texture` pointer and variables for the size of the camera's view and viewport. The viewport is the SFML concept, which defines the area of the screen the view will be displayed over. This will make full sense when we code the `.cpp` file in a moment. And to really make sure you grasp the concept of a viewport and how it is distinct from the view, we will discuss it in depth in the section *The SFML View class*, but let's add the code first, so we have more context for our discussion.

The `getTimeConnection` function returns a pointer to the `m_Time` variable, which will be called by the `LevelUpdate` class. This gives the `LevelUpdate` class the ability to change the `m_Time` variables value – which, as we will see, will then change the text in the top-left corner of the screen.

The `assemble` function is our usual overridden function that takes a `VertexArray`, a shared pointer to a generic `Update` instance, and the texture coordinates. The `draw` function is also overridden from the `Graphics` class and just needs the `VertexArray` to do its work.

Now, let's code all those functions to begin to understand them fully. Then we will dive a bit deeper into the SFML `View` class, as promised. Add the following to the `CameraGraphics.cpp` file.

```
#include "CameraGraphics.h"
#include "CameraUpdate.h"
CameraGraphics::CameraGraphics(
    RenderWindow* window, Texture* texture,
    Vector2f viewSize, FloatRect viewport)
{
    m_Window = window;
    m_Texture = texture;
    m_View.setSize(viewSize);
    m_View.setViewport(viewport);
    // The mini map viewport is less than 1
if (viewport.width < 1)
    {
        m_IsMiniMap = true;
    }
    else
    {
        // Only the full screen camera has the time text
        m_Font.loadFromFile("fonts/KOMIKAP_.ttf");
        m_Text.setFont(m_Font);
        m_Text.setFillColor(Color(255, 0, 0, 255));
        m_Text.setScale(0.2f, 0.2f);
    }
}
```

We just added the constructor for the `CameraGraphics` class. In this code, we begin by initializing the `RenderWindow` pointer and the `Texture` pointer. The view size is set by calling `setSize` and passing in the `viewSize` parameter,

and the viewport is set by calling `setViewport` and passing in `viewport`. Let's leave the `CameraGraphics` class for a few pages and take a closer look at the SFML `View` class.

The SFML View class

To expand a little on exactly what the viewport is, take a look at the next image, which contains a few examples.

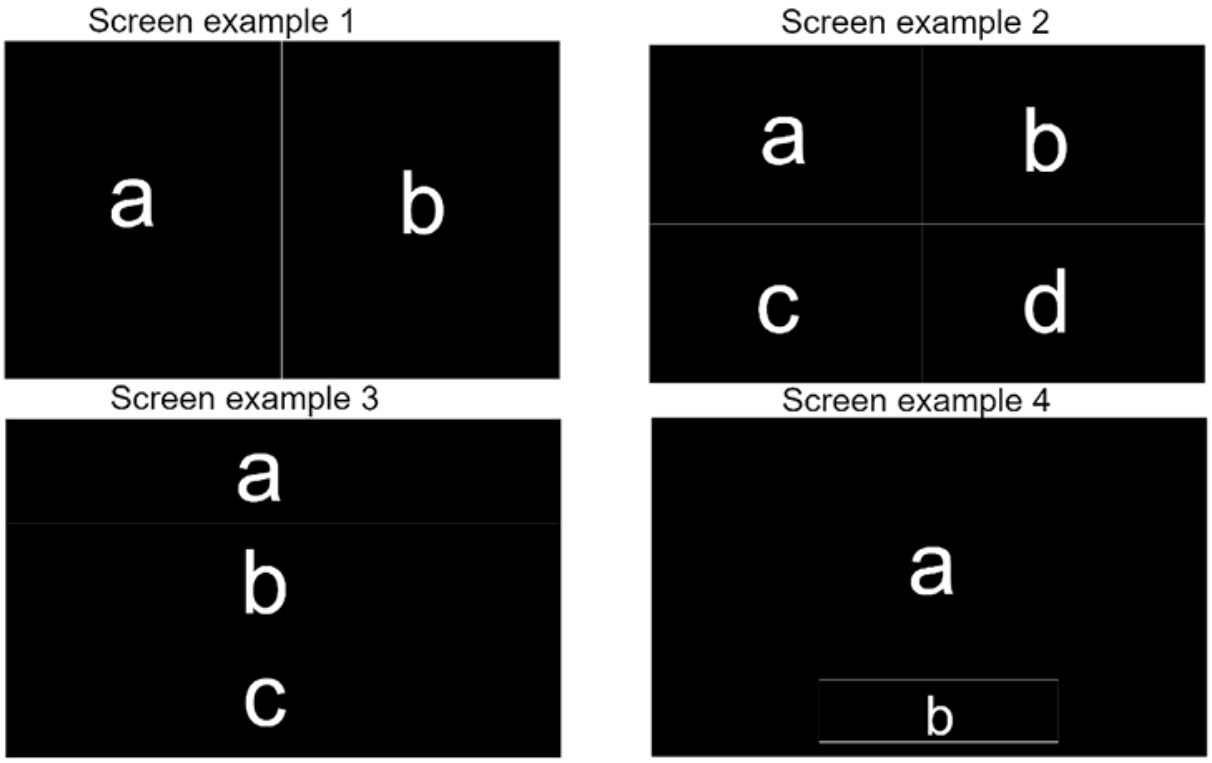


Figure 17.1: Viewports explained

In the preceding image, we see some examples of ways we can arrange the viewport of a `View` instance to control where and how much of the screen the call to draw will occupy. In all our previous projects, we have used the default values, which is the whole screen.

A viewport is defined by an SFML `FloatRect` with a `left` and `top` value defining the top-left corner and a `width` and `height` value defining how wide and high it is. The difference between the `viewport` and the `viewsize` (as set by the `setSize` function) is important.

The `setSize` function determines how many world units the view will show. Hoever, the viewport height and width determine what proportion of screen real estate that view will use. You could show a lot of the world in a very small viewport or a small amount of the world in a very large viewport depending on what suits the game. The viewports are defined in normalized coordinates. This means that the smallest possible value is `0` and the biggest is `1`.

Therefore, the default viewport that covers the entire screen has `top` and `left` values of `0` and `height` and `width` values of `1`. More examples will help clarify this.

Refer to **Screen example** `1` in the preceding image. The viewport labeled **a** would have the following values: left = `0`, top = `0`, width = `0.5`, and height = `1`. This is because the viewport starts at the far left and top (`0`) and goes for half the width of the screen (`0.5`) and the full height (`1`). Viewport **b** has values as follows: left = `0.5`, top = `0`, width = `0.5`, and height = `1`. This is because it starts horizontally in the middle of the screen (`0.5`), vertically at the top (`0`), and is half the width (`0.5`) and the full height (`1`).

Look at **Screen example 2** and spend a moment reconciling these values for viewports **a**, **b**, **c**, and **d**:

- Viewport **a**: left = `0`, top = `0`, width = `0.5`, height = `0.5`
- Viewport **b**: left = `0`.5, top = `0`, width = `0.5`, height = `0.5`
- Viewport **c**: left = `0`, top = `0.5`, width = `0.5`, height = `0.5`

- Viewport **d**: left = `0.5`, top = `0.5`, width = `0.5`, height = `0.5`

All the viewports have widths and heights of `0.5` because they all use half the height and half the width. The two viewports on the left of the screen have left values of `0`, the two viewports that start in the center of the screen have left values of `0.5`, etc.

Here are the values for the viewports in **Screen example 3**. I won't describe them all; just take the time to reconcile them based on what we have just discussed:

Viewport **a**: left = `0`, top = `0`, width = `1`, height = `0.33`

Viewport b: **left** = `0`, top = `0.33`, width = `1`, height = `0.33`

Viewport c: **left** = `0`, top = `0.66`, width = `1`, height = `0.33`

The final example, **Screen example 4**, is an approximate representation of the viewports in our game. Viewport a is the full screen starting in the top-left corner the same as the default: left = `0`, top = `0`, width = `1`, and height = `1`. Viewport b is the mini map/radar and has the following values: left = `0.2`, top = `0.8`, width = `0.6`, and height = `0.19`. We will see, when we add code to the `Factory` class that when we pass values for the viewport, we also take into account the screen resolution and the ratio of the horizontal-to-vertical resolution. Furthermore, because viewport **b** overlaps the real estate of viewport **a**, we must make sure the camera that uses viewport **b** is drawn second; otherwise, it will be covered by the camera that uses viewport **a**.

Let's get back to the code.

Coding the CameraGraphics class part 2

With all that, we have just learned that if the viewport is any less than 1 in either direction (in our game), it is not the full-screen camera. Therefore, if the code `if(viewport.width<1)` is `true`, then this is going to be the mini-map camera. Of course, we could pass the wrong values by accident, but the code assumes we get it right and, therefore, any width less than 1 is the mini map. Inside the `if` statement, `m_IsMiniMap` is set to `true`.

Inside the `else` statement that executes when it is the main camera, we load the font, set the font, color the font, and scale the font, just as we have done in all our other projects, albeit in a different part of the code. As mentioned previously, the mini map will not use or display the time and therefore doesn't need the `Font` or the `Text` instance.

Next, add this code, which is the `assemble` function, also into `CameraGraphics.cpp`.

```
void CameraGraphics::assemble(
    VertexArray& canvas,
    shared_ptr<Update> genericUpdate,
    IntRect texCoords)
{
    shared_ptr<CameraUpdate> cameraUpdate =
        static_pointer_cast<CameraUpdate>(genericUpdate);
    m_Position = cameraUpdate->getPositionPointer();
    m_VertexStartIndex = canvas.getVertexCount();
    canvas.resize(canvas.getVertexCount() + 4);
    const int uPos = texCoords.left;
    const int vPos = texCoords.top;
    const int texWidth = texCoords.width;
    const int texHeight = texCoords.height;
    canvas[m_VertexStartIndex].texCoords.x = uPos;
    canvas[m_VertexStartIndex].texCoords.y = vPos;
    canvas[m_VertexStartIndex + 1].texCoords.x =
```

```
            uPos + texWidth;
        canvas[m_VertexStartIndex + 1].texCoords.y =
            vPos;
        canvas[m_VertexStartIndex + 2].texCoords.x =
            uPos + texWidth;
        canvas[m_VertexStartIndex + 2].texCoords.y =
            vPos + texHeight;
        canvas[m_VertexStartIndex + 3].texCoords.x =
            uPos;
        canvas[m_VertexStartIndex + 3].texCoords.y =
            vPos + texHeight;
    }
```

In the `assemble` function that we just added, in the first line of code, we use `static_pointer_cast` to transform the generic `Update` shared pointer into a `CameraUpdate` shared pointer. Now the `CameraGraphics` class can call all the public functions of the `CameraUpdate` class. And the second line of code uses this possibility to initialize the `m_Position` pointer by calling the `getPositionPointer` function on the `CameraUpdate` class. Now we can always track where the camera should be drawn.

All the rest of the code in the `assemble` function saves the index of the quad for the camera and initializes all the texture coordinates related to that quad into the `VertexArray`. The texture coordinates all reconcile to a very light transparent rectangle to create a visual separation between the main camera and the mini-map camera.

Next, add this code, which is the `getTimeConnection` function, also into the `CameraGraphics.cpp` file.

```
float* CameraGraphics::getTimeConnection()
{
    return &m_Time;
}
```

This function, `getTimeConnection`, is short and sweet as it just returns the address of the `m_Time` variable. Once the `LevelUpdate` class has called this function and stored the result, it will be able to update the `m_Time` variable, which can then be updated at each frame in the `draw` function. Speaking of the `draw` function, we will code that next.

Finally, for the `CameraGraphics` class, also add this code into `CameraGraphics.cpp`.

```
void CameraGraphics::draw(VertexArray& canvas)
{
    m_View.setCenter(m_Position->getPosition());
    Vector2f startPosition;
    startPosition.x = m_View.getCenter().x -
        m_View.getSize().x / 2;
    startPosition.y = m_View.getCenter().y -
        m_View.getSize().y / 2;
    Vector2f scale;
    scale.x = m_View.getSize().x;
    scale.y = m_View.getSize().y;
    canvas[m_VertexStartIndex].position = startPosition;
    canvas[m_VertexStartIndex + 1].position =
        startPosition + Vector2f(scale.x, 0);
    canvas[m_VertexStartIndex + 2].position =
        startPosition + scale;
    canvas[m_VertexStartIndex + 3].position =
        startPosition + Vector2f(0, scale.y);
    if (m_IsMiniMap)
    {
        if (m_View.getSize().x <
            MAX_WIDTH && m_Position->width > 1)
        {
            m_View.zoom(m_Position->width);
        }
        else if (m_View.getSize().x >
            MIN_WIDTH && m_Position->width < 1)
        {
            m_View.zoom(m_Position->width);
        }
```

```
    }
    m_Window->setView(m_View);
    // Draw the time UI but only in the main camera
if (!m_IsMiniMap)
    {
        m_Text.setString(std::to_string(m_Time));
        m_Text.setPosition(
            m_Window->mapPixelToCoords(Vector2i(5, 5)));
        m_Window->draw(m_Text);
    }
    // Draw the main canvas
    m_Window->draw(canvas, m_Texture);
}
```

In the `draw` function above, we have some code that all cameras will use, some code that only the mini-map camera will use (`if (m_IsMiniMap)`), and some code that only the regular camera will use (`if (!m_IsMiniMap)`).

The start of the function has most of the code that both cameras will use. Here is a reminder.

```
m_View.setCenter(m_Position->getPosition());
Vector2f startPosition;
startPosition.x = m_View.getCenter().x -
m_View.getSize().x / 2;
    startPosition.y = m_View.getCenter().y -
        m_View.getSize().y / 2;
    Vector2f scale;
    scale.x = m_View.getSize().x;
    scale.y = m_View.getSize().y;
    canvas[m_VertexStartIndex].position = startPosition;
    canvas[m_VertexStartIndex + 1].position =
        startPosition + Vector2f(scale.x, 0);
    canvas[m_VertexStartIndex + 2].position =
        startPosition + scale;
    canvas[m_VertexStartIndex + 3].position =
        startPosition + Vector2f(0, scale.y);
```

In the preceding code, which both camera instances will use, the `startPosition Vector2f` variable is initialized using the `View` instances' size and center. Next, the `scale Vector2f` variable is initialized using the size of the `View` instance.

Finally (in the code above), the relevant vertices in the `VertexArray` are positioned using `startPosition` and `scale`.

The code that the mini map only uses is shown again next for clarity.

```
    if (m_IsMiniMap)
       {
           if (m_View.getSize().x <
               MAX_WIDTH && m_Position->width > 1)
           {
               m_View.zoom(m_Position->width);
           }
           else if (m_View.getSize().x >
               MIN_WIDTH && m_Position->width < 1)
           {
               m_View.zoom(m_Position->width);
           }
       }
```

In the code that only the mini map uses, shown above, there is an `if` section which wraps an `if-else if` structure. In the `if` section, which executes when `m_IsMiniMap` is `true`, the outer `if` section is entered. When the size of the `View` instance is less than the maximum allowed size and `m_Position.width` is less than 1, the view is zoomed. Remember from the `CameraUpdate` class that the amount of required zoom is stored in `m_Position.width`.

The inner `else-if` section executes when the minimum allowed zoom is exceeded and `m_Position.width` is less than 1. Inside the `else-if` structure,

the view is zoomed in.

Note that after this code, the view is set to the appropriate instance for both cameras. The default is always set to `1` at the start of the `update` function in the `CameraUpdate` class, which means the regular camera will never zoom, and if the scroll wheel is not used, neither camera will zoom.

```
m_Window->setView(m_View);
```

The next code is only used by the regular camera and is shown again here for clarity.

```
if (!m_IsMiniMap)
{
m_Text.setString(std::to_string(m_Time));
   m_Text.setPosition(
   m_Window->mapPixelToCoords(Vector2i(5, 5)));
   m_Window->draw(m_Text);
}
```

In the code that the regular camera uses shown above, the text in the top-left corner of the screen is configured with `setString` and `setPosition`. Finally, we call `draw` using the `RenderWindow` pointer.

The `draw` function is called once for every instance of the `CameraGraphics` class as opposed to dozens of times in our zombie game. This is because all the game objects are in the vertex array. This is much more efficient, which means our game would run on lower-spec PCs or we could add extra game objects before performance became an issue.

For completeness, here is the draw call again.

```
m_Window->draw(canvas, m_Texture);
```

To see our two new camera-related classes in action, we need to instantiate them in the `Factory` class, wrap them in `GameObject` instances, and add them to our vector that we iterate through in our game loop.

Adding camera instances to the game

We will have two cameras, one for the main view of the game and one for our mini map.

Open the `Factory.cpp` file. Add the following two highlighted `include` directives to the top of the file.

```
#include "Factory.h"
#include "LevelUpdate.h"
#include "PlayerGraphics.h"
#include "PlayerUpdate.h"
#include "InputDispatcher.h"
#include "CameraUpdate.h"
#include "CameraGraphics.h"
```

Now add the following code for the first camera. Add all the camera code after the code that handles the player, at the very end (but inside) of the `loadLevel` function. Note that the first couple of lines are for both cameras; we will get to the second camera next. The regular full-screen camera is added first; otherwise, it would hide the mini-map camera.

```
// For both the cameras
const float width = float(VideoMode::getDesktopMode().width);
const float height = float(VideoMode::getDesktopMode().height);
const float ratio = width / height;
// Main camera
GameObject camera;
```

```
shared_ptr<CameraUpdate> cameraUpdate =
make_shared<CameraUpdate>();
cameraUpdate->assemble(nullptr, playerUpdate);
camera.addComponent(cameraUpdate);
shared_ptr<CameraGraphics> cameraGraphics =
make_shared<CameraGraphics>(
m_Window, m_Texture,
Vector2f(CAM_VIEW_WIDTH, CAM_VIEW_WIDTH / ratio),
FloatRect(CAM_SCREEN_RATIO_LEFT, CAM_SCREEN_RATIO_TOP,
CAM_SCREEN_RATIO_WIDTH, CAM_SCREEN_RATIO_HEIGHT));
cameraGraphics->assemble(
canvas,
cameraUpdate,
IntRect(CAM_TEX_LEFT, CAM_TEX_TOP,
CAM_TEX_WIDTH, CAM_TEX_HEIGHT));
camera.addComponent(cameraGraphics);
gameObjects.push_back(camera);
levelUpdate->connectToCameraTime(
cameraGraphics->getTimeConnection());
// End Camera
```

In the preceding code, which is probably starting to feel familiar, first we add some code for the benefit of both instances. The `width`, `height`, and `ratio` variables are initialized based on the resolution of the screen that the game is running on. We will use these values on both cameras. Then we get to the code for the main camera.

First, we create a `GameObject` instance called `camera`. Next, we make a shared pointer `CameraUpdate` instance. Then we call the `assemble` function and pass in the `playerUpdate` shared pointer. Next, we add `cameraUpdate` to `camera` using the `addComponent` function.

Next up, we create a `CameraGraphics` shared pointer and call the constructor passing in the `RenderWindow`, the `Texture`, our constants for the camera size, and the viewport size. Next, we call the `assemble` function, passing in the `VertexArray`, the `cameraUpdate` instance (in its parent form `Update`), and

texture coordinates. We then add the `CameraGraphics` instance to the `GameObject` instance and add the `GameObject` instance to the `gameObjects` vector.

Next, we make the connection between the `LevelUpdate` instance and `CameraGraphics` instance by calling `connectToCameraTime` on `levelUpdate` and passing in the result from calling `getTimeConnection` on `cameraGraphics`.

Next, add the code for the camera that will be the mini map.

```
// MapCamera
GameObject mapCamera;
shared_ptr<CameraUpdate> mapCameraUpdate =
make_shared<CameraUpdate>();
mapCameraUpdate->assemble(nullptr, playerUpdate);
mapCamera.addComponent(mapCameraUpdate);
inputDispatcher.registerNewInputReceiver(
mapCameraUpdate->getInputReceiver());
shared_ptr<CameraGraphics> mapCameraGraphics =
make_shared<CameraGraphics>(
m_Window, m_Texture,
Vector2f(MAP_CAM_VIEW_WIDTH,
MAP_CAM_VIEW_HEIGHT / ratio),
FloatRect(MAP_CAM_SCREEN_RATIO_LEFT,
MAP_CAM_SCREEN_RATIO_TOP,
MAP_CAM_SCREEN_RATIO_WIDTH,
MAP_CAM_SCREEN_RATIO_HEIGHT));
mapCameraGraphics->assemble(canvas,
mapCameraUpdate,
IntRect(MAP_CAM_TEX_LEFT, MAP_CAM_TEX_TOP,
MAP_CAM_TEX_WIDTH, MAP_CAM_TEX_HEIGHT));
mapCamera.addComponent(mapCameraGraphics);
gameObjects.push_back(mapCamera);
// End Map Camera
```

In the preceding code, we use exactly the same techniques and code as we did for the first camera except the size and viewport are different. The size is bigger on the mini map because it shows a wider area, but the viewport is smaller because it is squished into a smaller area. This will be more-evident once we have more graphics in the game.

Running the game

Now our cameras are drawing the `VertexArray` to the screen and we can delete the extra line of code we added temporarily in the `main` function. Delete the following code from `Run.cpp`:

```
…
// Temporary code until next chapter
window.draw(canvas, factory.m_Texture);
…
```

Now we can run the game and see the cameras in action.

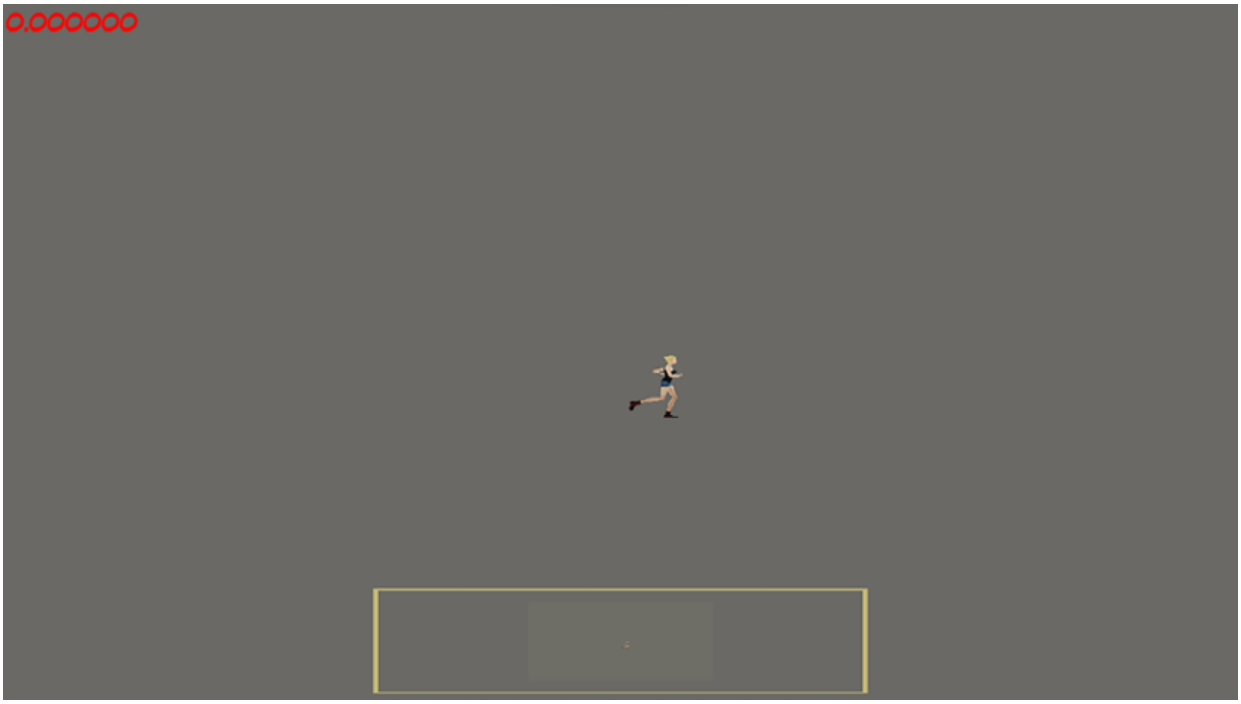


Figure 17.2: Cameras in action

In the preceding image, you can see the player correctly positioned and scaled.

Also, if you scroll the mouse wheel, you can see the mini map zooming in and out, although there isn't much to see in it yet.

In the next chapter, we will add the platforms, and then after that, in the same chapter, we can add animation and keyboard controls to the player. Remember, the `InputReceiver` instance in the `PlayerUpdate` class is already receiving all the events; we just need to respond to them.

Summary

In this chapter, we learned that it is much more efficient to minimize the number of draw calls and that we can achieve this by using a single `VertexArray` for all the entities in our game, although we did also have a

separate SFML `Text` instance on which we also called `draw`. Also, in this chapter, we coded our two cameras using our Entity Component pattern of an `Update` derived class and a `Graphics` derived class. Furthermore, we saw that these classes share data with each other to work effectively and that they also share data with the player-related classes.

We saw how we can add cameras in the factory, and by passing the required parameters like the appropriate `Update` and `Graphics` derived instances of other classes to the `assemble` functions, we can configure the cameras to behave as we like.

Now that our cameras are up and running, as well as our game logic that we added through the `LevelUpdate` class in *Chapter 16*, anything we add to the game now will bring instant gratification as we will be able to see it in action without delay. In the next chapter, we will be adding platforms to the game and animating and responding to keyboard inputs for the player.

18

Coding the Platforms, Player Animations, and Controls

In this chapter, we will code the platforms, player animations, and controls. In my opinion, we have done the hard work already and most of what follows has a much higher reward-to-effort ratio. Hopefully, this chapter will be interesting as we will see how the platforms will ground the player and enable them to run as well as see how we loop through the frames of animation to create a smooth running effect for the player.

We will do the following:

- Coding the platforms: You guessed it. Two classes are needed – one derived from `Update` and one from `Graphics`.
- Adding functionality to the player.
- Coding the `Animator` class.
- Coding the player animations: Add a smooth-running animation to the player.

The completed code for this chapter is in the `Run4` folder.

Coding the platforms

To get started, create the two classes we will need first in this chapter. They are `PlatformUpdate`, which extends `Update`, and `PlatformGraphics`, which extends `Graphics`. We already have the player-related classes, and we will add more code to them once we are done with the platforms. We will, however, need an `Animator` class that will control the animations of the player, and later in the project, it will also control the animations of the fireballs. Feel free to create an empty `Animator` class now or wait until we code it.

Coding the PlatformUpdate class

Most of what the platform will do is handle collisions with the player. If the player's feet are touching the top of the platform, then they should not be able to pass through. If the right side of the player touches the left side of the platform, then it should not pass through, and so on. This next image shows a representation of what the `PlatformUpdate` class will achieve.

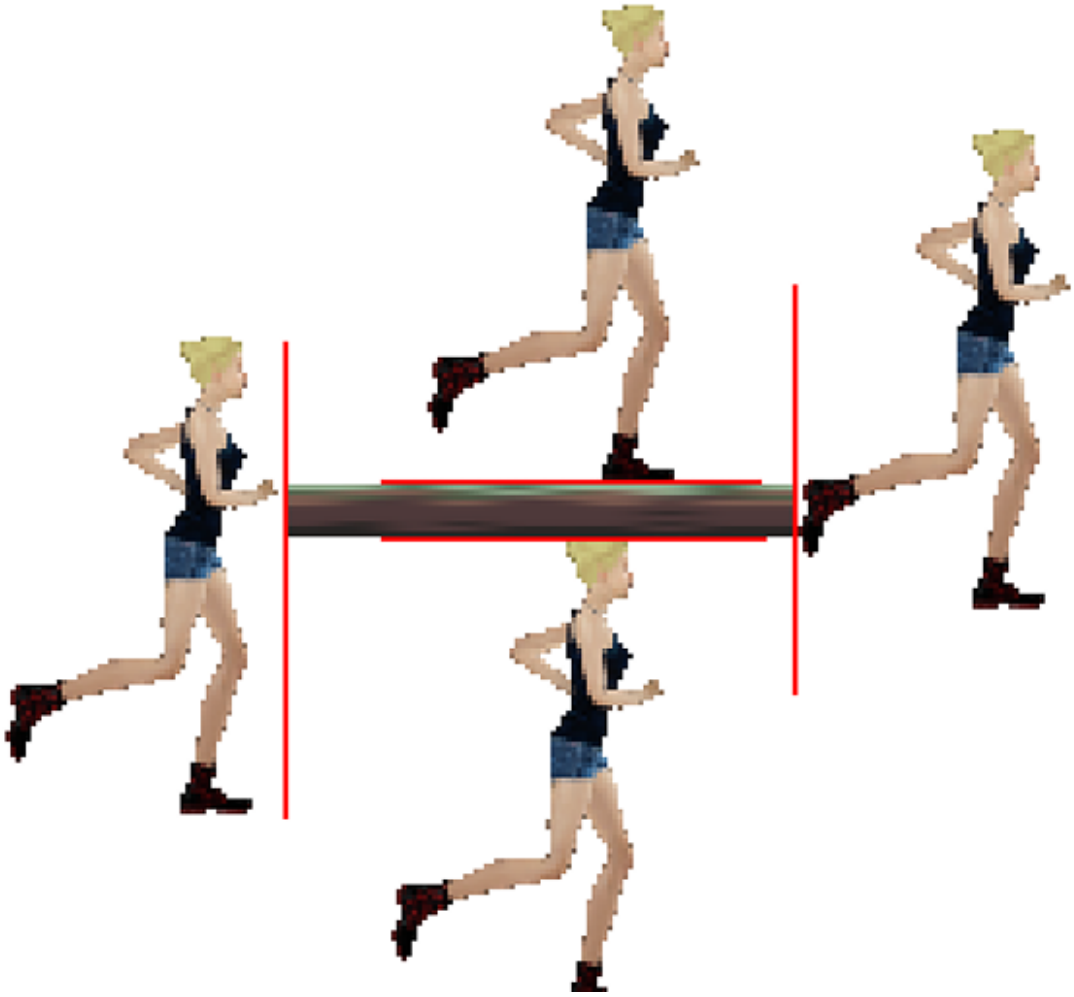


Figure 18.1: Platform colliding with the player

In the preceding image, the red lines indicate where the `PlatformUpdate` class will check for overlap and the position of each of the player images indicates the edge of the player involved in that intersection. When an intersection is detected, the player will be moved to the closest non-intersecting point (the lines), creating the effect of a solid object for the platforms.

Add the following into `PlatformUpdate.h`:

```
#pragma once
#include "Update.h"
```

```
#include "SFML/Graphics.hpp"
using namespace sf;
class PlatformUpdate :
public Update
{
private:
  FloatRect m_Position;
  FloatRect* m_PlayerPosition = nullptr;
  bool* m_PlayerIsGrounded = nullptr;
public:
  FloatRect* getPositionPointer();
  // From Update : Component
void update(float fps) override;
  void assemble(shared_ptr<LevelUpdate> levelUpdate,
    shared_ptr<PlayerUpdate> playerUpdate)
 override;
};
```

In the preceding code, we declare variables for position, a pointer for the player position, and a Boolean for whether or not the player is grounded. If the player is not grounded, it shouldn't be able to run. Also, where better to calculate whether the player is grounded than in the platform class?

In the `public` section, we have the `getPositionPointer` function, which returns the address of a `FloatRect` instance that will hold the position of the platform. This is how we will pass in the required manipulatable position to the `LevelUpdate` instance that we coded in *Chapter 15*.

Following on, we have the obligatory `update` and `assemble` functions. We have seen these definitions often. How we code them next will be most interesting.

Add the following code into `PlatformUpdate.cpp`.

```
#include "PlatformUpdate.h"
#include "PlayerUpdate.h"
FloatRect* PlatformUpdate::getPositionPointer()
```

```
{
    return &m_Position;
}
void PlatformUpdate::assemble(
    shared_ptr<LevelUpdate> levelUpdate,
    shared_ptr<PlayerUpdate> playerUpdate)
{
    //mPosition = position;
    m_PlayerPosition = playerUpdate->getPositionPointer();
    m_PlayerIsGrounded = playerUpdate->getGroundedPointer();
}
```

In the `getPositionPointer` function, we return the address of the `m_Position` variable.

Next, we code the `assemble` function that initializes `m_PlayerPosition` with the address of the player's position, which we obtain by using the `playerUpdate` shared pointer and calling the `getPositionPointer` function.

We then initialize the `m_PlayerIsGrounded` pointer with the address of the variable obtained from the `PlayerUpdate` class' `getGroundedPointer` function. Now, anything we do to `m_PlayerPosition` or `m_PLayerIsGrounded` will be instantly reflected directly back in the player-related classes.

Next, we will code the `update` function that executes once each iteration of the game loop.

Coding the update function for the PlatformUpdate class

To finish the `PlatformUpdate` class, add this code to `PlatformUpdate.cpp`:

```
void PlatformUpdate::update(float fps)
{
    if (m_Position.intersects(*m_PlayerPosition))
    {
```

```
        Vector2f playerFeet(m_PlayerPosition->left +
            m_PlayerPosition->width / 2,
            m_PlayerPosition->top +
            m_PlayerPosition->height);
        Vector2f playerRight(m_PlayerPosition->left +
            m_PlayerPosition->width,
            m_PlayerPosition->top +
            m_PlayerPosition->height / 2);
        Vector2f playerLeft(m_PlayerPosition->left,
            m_PlayerPosition->top +
            m_PlayerPosition->height / 2);
        Vector2f playerHead(m_PlayerPosition->left +
            m_PlayerPosition->width / 2,
            m_PlayerPosition->top);
        if (m_Position.contains(playerFeet))
        {
            if (playerFeet.y > m_Position.top)
            {
                m_PlayerPosition->top =
                    m_Position.top -
                    m_PlayerPosition->height;
                *m_PlayerIsGrounded = true;
            }
        }
        else if (m_Position.contains(playerRight))
        {
            m_PlayerPosition->left =
                m_Position.left - m_PlayerPosition->width;
        }
        else if (m_Position.contains(playerLeft))
        {
            m_PlayerPosition->left =
                m_Position.left + m_Position.width;
        }
        else if (m_Position.contains(playerHead))
        {
            m_PlayerPosition->top =
                m_Position.top + m_Position.height;
        }
    }
}
```

The `update` function is the one that does most of the work so let's break it down. All the code detects whether the player is colliding with the platform, which part of the player is colliding with the platform, and which part of the platform is colliding with the player. The first all-encompassing `if` statement detects whether there is any intersection whatsoever between the player and the platform.

```
if (m_Position.intersects(*m_PlayerPosition))
```

In the preceding code, an initial check is done. If there is an intersection anywhere between the player and the platform, then we need to do further tests to determine exactly what collision has occurred. If there is no intersection at all, then we can skip the rest of the code in the `update` function.

If there is some kind of intersection, then the next code defines the body parts of the player that we will test for to get the precise collision to handle:

```
Vector2f playerFeet(m_PlayerPosition->left +
m_PlayerPosition->width / 2,
            m_PlayerPosition->top +
            m_PlayerPosition->height);
Vector2f playerRight(m_PlayerPosition->left +
            m_PlayerPosition->width,
            m_PlayerPosition->top +
            m_PlayerPosition->height / 2);
Vector2f playerLeft(m_PlayerPosition->left,
            m_PlayerPosition->top +
            m_PlayerPosition->height / 2);
Vector2f playerHead(m_PlayerPosition->left +
            m_PlayerPosition->width / 2,
            m_PlayerPosition->top);
```

In the preceding code, four `Vector2f` instances are created: `playerFeet`, `playerRight`, `playerLeft`, and `playerHead`. The instances are initialized by calling the `Vector2f` constructor and passing in the appropriate values from the `m_PlayerPosition` pointer, which points to the `m_Position` variable in the `PlayerUpdate` class.

The next `if` and three `else-if` statements handle what to do in each collision case; head, left, right, and feet respectively:

```
if (m_Position.contains(playerFeet))
{
if (playerFeet.y > m_Position.top)
   {
       m_PlayerPosition->top =
          m_Position.top -
        m_PlayerPosition->height;
       *m_PlayerIsGrounded = true;
   }
}
        else if (m_Position.contains(playerRight))
        {
            m_PlayerPosition->left =
                m_Position.left - m_PlayerPosition->width;
        }
        else if (m_Position.contains(playerLeft))
        {
            m_PlayerPosition->left =
                m_Position.left + m_Position.width;
        }
        else if (m_Position.contains(playerHead))
        {
            m_PlayerPosition->top =
                m_Position.top + m_Position.height;
        }
```

In the preceding code, when the feet intersect, they are realigned with the top of the platform; when the right intersects, it is realigned with the left of the

platform; when the left intersects, it is realigned with the right of the platform; and when the head intersects, it is realigned with the underside of the platform. Check back to the preceding image to get a visual representation of how this makes a platform into a solid impenetrable object to the player.

Other game entities like fireballs will pass through platforms because this is fine for our game, but it would be trivial to get pointers to all the fireballs and handle collision for them or any other entity in the game should we wish to.

Now, we just need to give our platforms their appearance.

Coding the PlatformGraphics class

Next, we can code the `PlatformGraphics` class, which will represent the data in the `PlatformUpdate` class visually. Add the following into `PlatformGraphics.h`:

```
#pragma once
#include "Graphics.h"
#include "SFML/Graphics.hpp"
using namespace sf;
class PlatformGraphics : public Graphics
{
private:
  FloatRect* m_Position = nullptr;
  int m_VertexStartIndex = -1;
public:
  //From Graphics : Component
void draw(VertexArray& canvas) override;
  void assemble(VertexArray& canvas,
    shared_ptr<Update> genericUpdate,
    IntRect texCoords) override;
};
```

In the preceding code, in the private section, there is a `FloatRect` pointer called `m_Position`, which will point to the `PlatformUpdate` class and hold the current position of the platform. Remember that the platforms will be regularly repositioned by the `LevelUpdate` class, which holds a vector of all the platform positions.

The `m_VertexStartIndex` integer fulfills the usual role of remembering the position in the `VertexArray` that the quad for this entity begins.

In the `public` section, we just have the usual two functions for classes that extend the `Graphics` class. They are `draw`, which takes a reference to the `VertexArray`, and `assemble`, which is used to prepare each platform and is called from the factory for each platform instance. We will code this factory-related code once we have completed this class.

Now, we can code the definitions of our two overridden functions. Add the following into `PlatformGraphics.cpp`:

```
#include "PlatformGraphics.h"
#include "PlatformUpdate.h"
void PlatformGraphics::draw(VertexArray& canvas)
{
    const Vector2f& position = m_Position->getPosition();
    const Vector2f& scale = m_Position->getSize();
    canvas[m_VertexStartIndex].position = position;
    canvas[m_VertexStartIndex + 1].position =
        position + Vector2f(scale.x, 0);
    canvas[m_VertexStartIndex + 2].position =
        position + scale;
    canvas[m_VertexStartIndex + 3].position =
        position + Vector2f(0, scale.y);
}
```

In the `draw` function we just coded, all we need to do is initialize the appropriate indexes of the `VertexArray` with the values pointed to by

`m_Position`. Although in most frames of the game, the platform will not move, we initialize the `VertexArray` because eventually, it will move.

If we had thousands of platforms, we could optimize this by adding a Boolean to the `PlatformUpdate` class, which indicates whether the platform has moved this frame, and only execute the preceding code when it has moved. This optimization won't be necessary in our game. I just thought you might like to be aware of the possibility.

To finish the `PlatformGraphics` class, also add the `assemble` function into `PlatformGraphics.cpp`:

```
void PlatformGraphics::assemble(VertexArray& canvas,
    shared_ptr<Update> genericUpdate,
    IntRect texCoords)
{
    shared_ptr<PlatformUpdate> platformUpdate =
        static_pointer_cast<PlatformUpdate>(genericUpdate);
    m_Position = platformUpdate->getPositionPointer();
    m_VertexStartIndex = canvas.getVertexCount();
    canvas.resize(canvas.getVertexCount() + 4);
    const int uPos = texCoords.left;
    const int vPos = texCoords.top;
    const int texWidth = texCoords.width;
    const int texHeight = texCoords.height;
    canvas[m_VertexStartIndex].texCoords.x = uPos;
    canvas[m_VertexStartIndex].texCoords.y = vPos;
    canvas[m_VertexStartIndex + 1].texCoords.x =
        uPos + texWidth;
    canvas[m_VertexStartIndex + 1].texCoords.y =
        vPos;
    canvas[m_VertexStartIndex + 2].texCoords.x =
        uPos + texWidth;
    canvas[m_VertexStartIndex + 2].texCoords.y =
        vPos + texHeight;
```

```
        canvas[m_VertexStartIndex + 3].texCoords.x =
            uPos;
        canvas[m_VertexStartIndex + 3].texCoords.y =
            vPos + texHeight;
    }
```

In the preceding `assemble` function, the generic `Update` instance passed in as a parameter is cast to be a `PlatformUpdate` instance. Now, the `m_Position` pointer can be initialized by calling `platformUpdate->getPositionPointer()`.

Moving on, the start index of the quad is ascertained and stored and the `VertexArray` is resized by adding four more vertices.

Finally, the texture coordinates are initialized into the appropriate positions of the `VertexArray`. Now, our class is ready to use.

Building some platforms in the factory

Add the following into `Factory.cpp` to spawn some platforms. First, add two new `include` directives, as shown next:

```
#include "PlatformUpdate.h"
#include "PlatformGraphics.h"
```

Now, add this code before the code we added for the cameras but after the code for the player, as shown by the highlighted code that follows (the preexisting line of code is highlighted):

```
// Make the LevelUpdate aware of the player
levelUpdate->assemble(nullptr, playerUpdate);
// For the platforms
for (int i = 0; i < 8; ++i)
{
```

```
    GameObject platform;
    shared_ptr<PlatformUpdate> platformUpdate =
      make_shared<PlatformUpdate>();
    platformUpdate->assemble(nullptr, playerUpdate);
    platform.addComponent(platformUpdate);
    shared_ptr<PlatformGraphics> platformGraphics =
      make_shared<PlatformGraphics>();
    platformGraphics->assemble(
      canvas, platformUpdate,
      IntRect(PLATFORM_TEX_LEFT, PLATFORM_TEX_TOP,
      PLATFORM_TEX_WIDTH, PLATFORM_TEX_HEIGHT));
    platform.addComponent(platformGraphics);
    gameObjects.push_back(platform);
    levelUpdate->addPlatformPosition(
      platformUpdate->getPositionPointer());
  }
  // End platforms
```

In the preceding code that we added to `Factory.cpp`, the code is wrapped in a `for` loop that executes eight times. You can experiment with more or fewer platforms but eight seems to work quite nicely. What follows happens each iteration through the `for` loop.

First, we create a new `GameObject` instance called `platform` and a `PlatformUpdate` instance shared pointer called `platformUpdate`. We then call `assemble` on `platformUpdate` and pass in the `playerUpdate` instance. Next, we call `addComponent` to add `platformUpdate` to `platform`.

Moving on, we make a `PlatformGraphics` shared pointer instance. Then, as with all our `Graphics` derived classes, we call `assemble` and pass in the `VertexArray`, the `platformUpdate` instance (as a generic `Update` instance), and the texture coordinates.

Now, we add the `platformGraphics` instance to the `GameObject` (`platform`) and call `push_back` on `gameObjects` to add `platform` to the vector of `GameObject` instances.

Finally, we use `levelUpdate` and call `addPlatformPosition` and pass in the result of calling `platformUpdate->getPositionPointer()`, which has the effect of allowing the `LevelUpdate` class to manipulate the position of the platform we just created. The `for` loop ensures this is repeated a further seven times.

Let's see how we have progressed by running the game.

Running the game

Temporarily change one line of code in the `LevelUpdate.h` file as shown next:

```
bool m_IsPaused = false;
```

Changing `m_isPaused` to `false` will let the platforms spawn. Now, run the game.

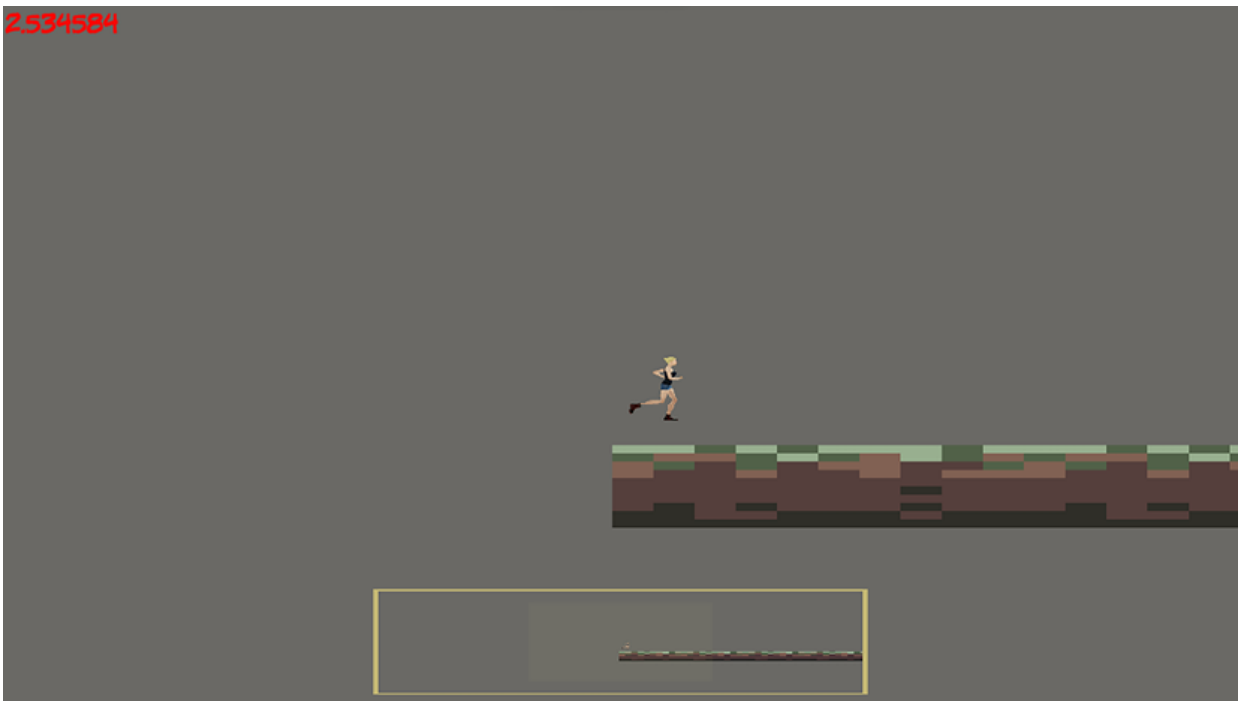


Figure 18.2: See the platforms

Notice that the timer in the top left is running and that you can see the platforms disappearing behind the player as they reappear in front.

Change `m_IsPaused` back to `true`. Let's bring the player character to life.

Adding functionality to the player

Of course, the player can't do anything yet. We will change that in two ways.

We will respond to the keyboard inputs read by our `InputReceiver`. This will take place in the `handleInput` function. At this point, the player will be able to move. Once we have done this, we will move on to animating the movement.

Coding the player controls

`PlayerUpdate.h` already has all the variables we need; we just need to utilize them in the `PlayerUpdate.cpp` file. Add the full code of the `handleInput` function to `PlayerUpdate.cpp`. Here is the function in its entirety:

```
void PlayerUpdate::handleInput()
{

        if (event.type == Event::KeyPressed)
        {
            if (event.key.code == Keyboard::D)
            {
                m_RightIsHeldDown = true;
            }
            if (event.key.code == Keyboard::A)
            {
                m_LeftIsHeldDown = true;
            }
```

```
            if (event.key.code == Keyboard::W)
            {
                m_BoostIsHeldDown = true;
            }
            if (event.key.code == Keyboard::Space)
            {
                m_SpaceHeldDown = true;
            }
        }
        if (event.type == Event::KeyReleased)
        {
            if (event.key.code == Keyboard::D)
            {
                m_RightIsHeldDown = false;
            }
            if (event.key.code == Keyboard::A)
            {
                m_LeftIsHeldDown = false;
            }
            if (event.key.code == Keyboard::W)
            {
                m_BoostIsHeldDown = false;
            }
            if (event.key.code == Keyboard::Space)
            {
                m_SpaceHeldDown = false;
            }
        }
    }
    m_InputReceiver.clearEvents();
}
```

In the preceding code, there are two `if` statements. The first executes when a keyboard key has been pressed and the other executes when a keyboard key has been released. For each of these possibilities, we respond to the *W*, *A*, and *D* keys and the spacebar. For each combination of key and movement (up and down), we set a Boolean variable. It will now be possible to respond to the state of all the keys from within the `update` function. Remember that

the `update` function was where this `handleInput` function was called from. Therefore, we will respond to the Boolean variables we have just set after the call to `handleInput`.

Next, we will add the code for the `update` function to `PlayerUpdate.cpp` but we will do so in small sections. Here is the `update` function broken into manageable pieces. Notice the one existing line of code moves. It will be simplest to start this function from scratch, as shown next. It is a long function but I have broken it into sections to code and explain. It might be worth adding all the sections first and then coming back to examine each in turn – whatever you are most comfortable doing. If there is any doubt about the order or position of the code in the `update` function, refer to the code file in the `PlayerUpdate.cpp` file in the `Run4` folder.

All the code takes place inside this structure. Add the following code first:

```
void PlayerUpdate::update(float timeTakenThisFrame)
{
    if (!*m_IsPaused)
    {
    // All the rest of the code
 }
}
```

In the preceding code, we first test whether the game is paused. If the game is paused, none of the code in the `update` function will execute.

All the rest of the code goes under this comment: `// All the rest of the code`.

Next, add this code:

```
m_Position.top += m_Gravity *
      timeTakenThisFrame;
```

```
handleInput();
if (m_IsGrounded)
{
        if (m_RightIsHeldDown)
        {
         m_Position.left +=
             timeTakenThisFrame * m_RunSpeed;
       }
        if (m_LeftIsHeldDown)
        {
           m_Position.left -=
             timeTakenThisFrame * m_RunSpeed;
         }
}
```

Notice in the preceding code that we are testing, for the values of the Boolean variables we have previously set in the `handleInput` function and responding to the changing the values held in `m_Position`.

The first line of code always executes (apart from when the game is paused) and pushes the player down in the world by the strength of gravity (`m_Gravity`) multiplied by how long the main game loop took to execute (`m_TimeTakenThisFrame`).

Now, the `handleInput` function is called to set all the Booleans. The next `if` statement checks whether the player is grounded. The reason for this is that we only want to respond to the player running left or right if they are grounded because you can't run in mid-air. If the player is grounded and the player is holding left or right (*A* or *D*), then `m_Position` is moved left or right accordingly based on the time taken by the game loop and the speed of the player (`m_RunSpeed`).

To handle some more movement, add the following code next:

```
if (m_BoostIsHeldDown)
   {
       m_Position.top -=
         timeTakenThisFrame * m_BoostSpeed;
       if (m_RightIsHeldDown)
       {
         m_Position.left +=
         timeTakenThisFrame * m_RunSpeed / 4;
      }
      if (m_LeftIsHeldDown)
      {
         m_Position.left -=
          timeTakenThisFrame * m_RunSpeed / 4;
      }
   }
```

The preceding code only executes when the boost button (*W*) is held down. If the boost button is held down, the player is moved up in the world based on the boost power (`m_BoostSpeed`) and the time the frame took to execute. In addition to moving up in the world, if *A* or *D* is held down, the player moves left and right as if the player is grounded and running. However, notice that the movement amount is divided by `4`. This is to make moving left and right while boosting slow and inconvenient. This is so that simply boosting away to the right as a method of getting a good score will not work. Boosting is just for emergencies, like falling off a platform and hovering above an incoming fireball.

Next, add this code:

```
// Handle Jumping
if (m_SpaceHeldDown && !m_InJump && m_IsGrounded)
{
SoundEngine::playJump();
m_InJump = true;
m_JumpClock.restart();
}
```

```
if (!m_SpaceHeldDown)
{
//mInJump = false;
}
```

The preceding code handles whether the player is trying to jump by testing to see whether the player has pressed the *spacebar* at the same time as being grounded. If they have, the jump sound is played, `m_InJump` is set to `true`, and the clock (`m_JumpClock`) is restarted to begin measuring how long the player has been in the jump.

Moving to the final part of the `update` function, add this code:

```
if (m_InJump)
{
if (m_JumpClock.getElapsedTime().asSeconds() <
m_JumpDuration / 2)
{
     // Going up
m_Position.top -= m_JumpSpeed *
         timeTakenThisFrame;
    }
else
   {
// Going down
     m_Position.top +=
m_JumpSpeed * timeTakenThisFrame;
}
if (m_JumpClock.getElapsedTime().asSeconds() >
      m_JumpDuration)
   {
m_InJump = false;
    }
if (m_RightIsHeldDown)
   {
     m_Position.left +=
         timeTakenThisFrame * m_RunSpeed;
       }
       if (m_LeftIsHeldDown)
```

```
        {
           m_Position.left -=
            timeTakenThisFrame * m_RunSpeed;
        }
}// End if(m_InJump)
   m_IsGrounded = false;
```

All the preceding code controls what happens when the player is in the jumping state, as determined by the Boolean `m_InJump`. The first `if` statement after it has been determined that the player is jumping detects whether the jump phase has passed the midway point with this code:

```
if (m_JumpClock.getElapsedTime().asSeconds() <
    m_JumpDuration / 2)
```

If it hasn't passed the midway point, the player is moved upwards (in the `if` block); if it has passed the midway point, the code in the `else` block moves the player downwards.

Next, the following code decides whether it is time to end the jump. Here it is again:

```
if (m_JumpClock.getElapsedTime().asSeconds() >
      m_JumpDuration)
{
m_InJump = false;
}
```

Finally, for the jump part of the code, the *left* and *right* keys are tested and the player is moved left or right if the appropriate key is held down. Notice the player moves at the same speed while in a jump as they do while running. This is almost scientifically accurate, but the main point is it is always preferable to run and jump rather than boost whenever possible.

Running the game

At this point, you could change one line of code in the `LevelUpdate.h` (if it isn't already changed) file as shown next:

```
bool m_IsPaused = false;
```

This would enable you to see the player skating across the level, jumping and boosting but without any animations.

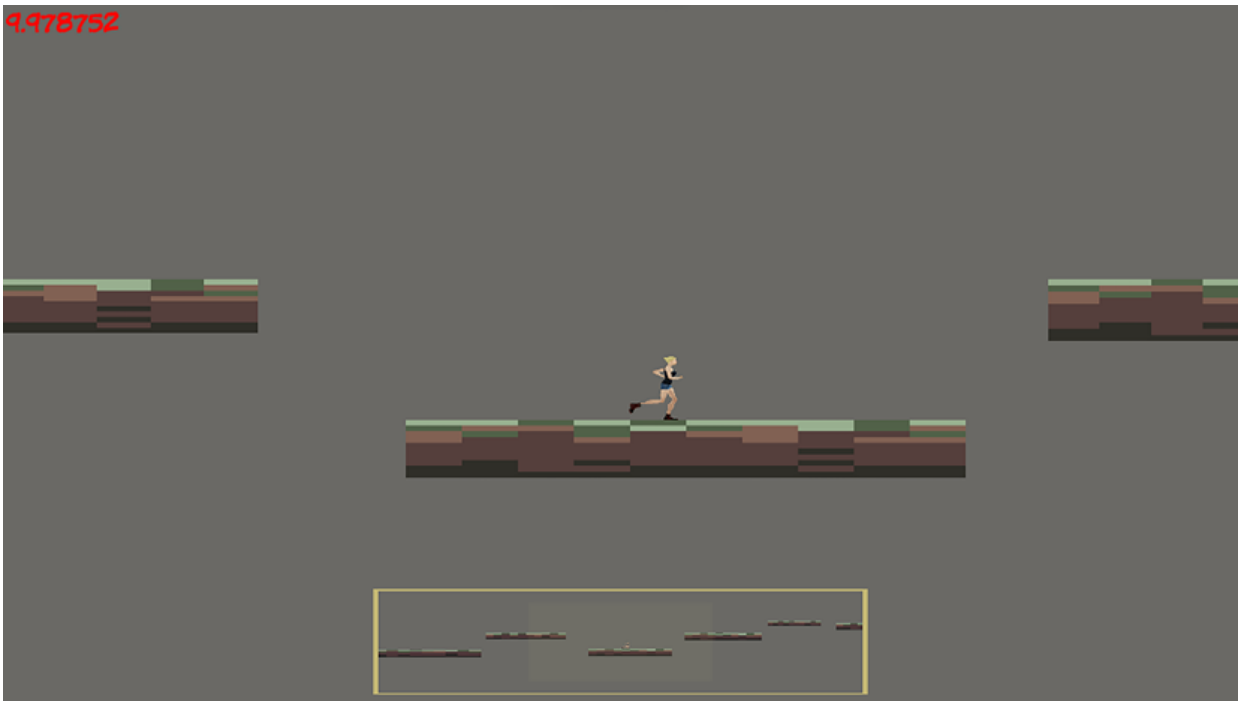


Figure 18.3: Unanimated player moving

Next, we will animate the player because this isn't a skating game. We will do this in two stages. First, we will create a class that picks a frame of animation (a set of texture coordinates) from the texture atlas and then we will add code to the `PlayerGraphics` class that will use an instance of our `Animator` class as well as some more code.

Coding the Animator class

First, let's get the `Animator` class up and running. The `Animator` class will also be used by the `FireballGraphics` class and the `RainGraphics` class. It could be used by any class that wants to loop through set frames of animations. It can be configured for any set of animations as long as they are evenly sized, evenly spaced, and have the same vertical coordinates. The `Animator` class can be configured by the code that uses it to reverse the animation order, which is useful when the player runs in the opposite direction as we will soon see. Reversing the animation can also produce a moon-walking effect but I will leave the reader to explore this possibility. The frame rate per second and the number of frames can also be determined at run time.

Create a class called `Animator` (if you haven't already).

Add this code to `Animator.h`:

```
#pragma once
#include<SFML/Graphics.hpp>
using namespace sf;
class Animator
{
private:
  IntRect m_SourceRect;
  int m_LeftOffset;
  int m_FrameCount;
  int m_CurrentFrame;
  int m_FramePeriod;
  int m_FrameWidth;
  int m_FPS = 12;
  Clock m_Clock;
public:
  Animator(
    int leftOffset, int topOffset,
    int frameCount,
```

```
      int textureWidth,
      int textureHeight,
      int fps);
    IntRect* getCurrentFrame(bool reversed);
  };
```

Let's step through the `Animator.h` file we have just coded. The `IntRect` instance called `m_SourceRect` will hold the integer coordinates of the current frame of animation in the texture atlas.

The `m_LeftOffset` variable is used to keep track of the horizontal value that defines the left-hand side of the current frame of animation. We will soon see in our code that we add `m_FrameWidth` to this value to move to the next set of texture coordinates.

The integer called `m_FrameCount` stores the number of frames that the animation sequence has. The `m_CurrentFrame` variable is the frame number of the current frame to be drawn.

The integer called `m_FramePeriod` is the duration of each frame of animation. This is calculated with 1 divided by the number of frames.

The integer `m_FrameWidth` holds the width of a frame of animation. This never changes for a given animation set.

The `m_FPS` variable will hold the number of frames that will be animated in each second. The `Clock` instance `m_Clock` keeps the time for the animation frame rate.

The `Animator` constructor has the following parameters: `int leftOffset`, `int topOffset`, `int frameCount`, `int textureWidth`, `int textureHeight`, and `int fps`. These all correspond to one of the member variables and will be assigned accordingly.

The `getCurrentFrame` function is the function that will do the work of calculating the current frame to draw and return the coordinates of the texture in the texture atlas in an `IntRect` pointer. The Boolean `reversed` parameter tells the function to calculate the frames moving right to left in the texture atlas when `reversed` is set to `true`.

Let's code the implementations in `Animator.cpp`. First, add the constructor implementation code as follows to `Animator.cpp`:

```
#include "Animator.h"
Animator::Animator(
  int leftOffset, int topOffset,
  int frameCount,
  int textureWidth,
  int textureHeight,
  int fps)
{
  m_LeftOffset = leftOffset;
  m_CurrentFrame = 0;
  m_FrameCount = frameCount;
  m_FrameWidth = (float)textureWidth
    / m_FrameCount;
  m_SourceRect.left = leftOffset;
  m_SourceRect.top = topOffset;
  m_SourceRect.width = m_FrameWidth;
  m_SourceRect.height = textureHeight;
  m_FPS = fps;
  m_FramePeriod = 1000 / m_FPS;
  m_Clock.restart();
}
```

In the preceding code for the `Animator` constructor, the values for `leftOffset`, `currentFrame`, and `frameCount` are initialized. The frame width is calculated by dividing the texture width by the frame count. The starting value for the `IntRect` (`m_SourceRect`), which holds the current texture coordinates, is initialized using left offset, top offset, frame width, and

texture height. This makes sense when you consider that we know all the frames of animation are in an equally spaced row.

Next, add the `getCurrentFrame` function to `Animator.cpp`:

```
IntRect* Animator::getCurrentFrame(bool reversed)
{
  // Reversed adds 1 to the frame number
// when drawing the texture reversed.
// This works because reversed
// (flipped horizontally) textures
// are drawn pixels right to left
if (m_Clock.getElapsedTime().asMilliseconds()
    > m_FramePeriod)
  {
    m_CurrentFrame++;
    if (m_CurrentFrame >= m_FrameCount + reversed)
    {
      m_CurrentFrame = 0 + reversed;
    }
    m_Clock.restart();
  }
  m_SourceRect.left = m_LeftOffset + m_CurrentFrame
    * m_FrameWidth;
  return &m_SourceRect;
}
```

In the `getCurrentFrame` function, the first `if` statement checks whether it is time to advance to the next frame. If it is `m_CurrentFrame` is incremented. The next `if` statement makes sure we haven't gone past the last frame. If we have the frame, it is set to zero inside the `if` block. The penultimate line of code initializes the texture coordinates inside `m_SourceRect` and then `m_SourceRect` is returned to the calling code. We will now move on to coding the `PlayerGraphics` class that calls the functions we have just coded.

Coding the player animations

In this section, we will get to use the `Animator` class we have just coded. We will obviously be using the `getCurrentFrame` function but additionally, we will be referring to the individual frames in the texture atlas like the player boosting, shown again next.

Figure 18.4: Player boosting

Furthermore, we have seen that our `Animator` class can reverse the order the frames are animated, but we also need to flip the textures horizontally whenever the player is facing to the left. In this code, we will see how to detect that they need to be flipped as well as flipping them. For example, the preceding image will sometimes need to be drawn like this:

Figure 18.5: Player boosting flipped

Achieving this is very quick and easy and we will see it in action for all player images soon. The `PlayerGraphics.h` file has everything we need in it; just uncomment the following code:

```
// We will come back to this soon
class Animator;
And the following code:
// We will come back to this soon
Animator* m_Animator;
```

What we have just done is added an `Animator` instance and a forward declaration for the `Animator` class.

Now, we just need to add some code to `PlayerGraphics.cpp`. First, add an `include` directive to `PlayerGraphics.cpp`:

```
#include "Animator.h"
```

Much of the original code we put in the assemble function is no longer needed now we have our `Animator`, so you can replace the `PlayerGraphics assemble` function as follows:

```
void PlayerGraphics::assemble(VertexArray& canvas,
    shared_ptr<Update> genericUpdate,
    IntRect texCoords)
{
    m_PlayerUpdate = static_pointer_cast<PlayerUpdate>(genericUp
    m_Position = m_PlayerUpdate->getPositionPointer();
    m_Animator = new Animator(
        texCoords.left,
        texCoords.top,
        6,// 6 frames
        texCoords.width * 6,
        texCoords.height,
        12);// FPS
```

```
    // Get the first frame of animation
        m_SectionToDraw = m_Animator->getCurrentFrame(false);
        m_StandingStillSectionToDraw = m_Animator->getCurrentFrame(f
        m_VertexStartIndex = canvas.getVertexCount();
        canvas.resize(canvas.getVertexCount() + 4);
    }
```

In the updated `assemble` function, we get a pointer to the position of the player in the `PlayerUpdate` class by casting the `Update` instance to a `PlayerUpdate` instance and calling the `getPositionPointer` function.

Next, we call `new` to initialize our `Animator` instance and pass in the required parameters, which include specifying the left and top texture coordinates, `6` frames in total, the total width and height, and the rate at `12` frames per second. The `Animator` class we coded previously will use this data to supply the correct frame of animation whenever we call the `getCurrentFrame` function. We could have made `getCurrentFrame` a function of the `PlayerGraphics` class but then we wouldn't be able to use it so easily on our fireballs and rain effects. As we have an `Animator` class, we can reuse it as often as we like and will do so for our fireballs and rain effects.

The next line of code initializes the `m_SectionToDraw IntRect` by calling the `getCurrentFrame` function. We then initialize `m_StandingStillSectionToDraw` by calling the same function again. This first frame is the one we will use when the player is static.

Finally, in the `assemble` function, the starting vertex of the quad is saved by calling `canvas.getVertexCount` and subtracting `1` from the returned value. Then, we can expand the `VertexArray` by calling `canvas.resize`.

The `draw` function is totally transformed, so we will replace it entirely with the following code into `PlayerGraphics.cpp`.

The `draw` function is long but it is not particularly helpful to break it up into more functions so I will just break it into sections to explain it. I recommend copying and pasting the entire `draw` function from the `PlayerGraphics.cpp` file in the `Run4` folder if you have any trouble interpreting the position or structure of any of the code that follows. The code isn't especially complex but there are just a lot of possibilities that we need to consider for the player to be drawn. For example: is the player moving, jumping, boosting, standing still, or facing left or right? All these options and different combinations of the options change how we want to draw the player. Get the code working, run it, play with it, and then come back here to learn how it works.

The first part of the `draw` function is as follows:

```
void PlayerGraphics::draw(VertexArray& canvas)
{
    const Vector2f& position =
        m_Position->getPosition();
    const Vector2f& scale =
        m_Position->getSize();
    canvas[m_VertexStartIndex].position =
        position;
    canvas[m_VertexStartIndex + 1].position =
        position + Vector2f(scale.x, 0);
    canvas[m_VertexStartIndex + 2].position =
        position + scale;
    canvas[m_VertexStartIndex + 3].position =
        position + Vector2f(0, scale.y);
    if (m_PlayerUpdate->m_RightIsHeldDown &&
        !m_PlayerUpdate->m_InJump &&
        !m_PlayerUpdate->m_BoostIsHeldDown &&
        m_PlayerUpdate->m_IsGrounded)
    {
        m_SectionToDraw = m_Animator->getCurrentFrame(false);
    }
    if (m_PlayerUpdate->m_LeftIsHeldDown &&
        !m_PlayerUpdate->m_InJump &&
        !m_PlayerUpdate->m_BoostIsHeldDown &&
```

```
            m_PlayerUpdate->m_IsGrounded)
        {
            m_SectionToDraw = m_Animator->getCurrentFrame(true);
    // reversed
        }
        else
        {
            // Test the players facing position
    // in case it changed while jumping or boosting
    // This value is used in the final animation option
    if (m_PlayerUpdate->m_LeftIsHeldDown)
            {
                m_LastFacingRight = false;
            }
            else
            {
                m_LastFacingRight = true;
            }
        }
```

In the preceding section, we position the vertices, decide whether to get the frame to the left or right of the previous frame, and set the `m_LastFacingRight` variable. In the next few sections, we will use the appropriate frame and position it on the `VertexArray` instance.

Add the following to the `draw` function:

```
    const int uPos = m_SectionToDraw->left;
       const int vPos = m_SectionToDraw->top;
       const int texWidth = m_SectionToDraw->width;
       const int texHeight = m_SectionToDraw->height;
       if (m_PlayerUpdate->m_RightIsHeldDown &&
           !m_PlayerUpdate->m_InJump &&
           !m_PlayerUpdate->m_BoostIsHeldDown)
       {
           canvas[m_VertexStartIndex].texCoords.x
               = uPos;
           canvas[m_VertexStartIndex].texCoords.y
               = vPos;
```

```
            canvas[m_VertexStartIndex + 1].texCoords.x
                = uPos + texWidth;
            canvas[m_VertexStartIndex + 1].texCoords.y
                = vPos;
            canvas[m_VertexStartIndex + 2].texCoords.x
                = uPos + texWidth;
            canvas[m_VertexStartIndex + 2].texCoords.y
                = vPos + texHeight;
            canvas[m_VertexStartIndex + 3].texCoords.x
                = uPos;
            canvas[m_VertexStartIndex + 3].texCoords.y
                = vPos + texHeight;
        }
```

In the preceding `draw` code section, we test whether the player is holding right down, not jumping, and not boosting; in other words, just running right. If this is the case, we just want to keep looping through our frames of animation facing the normal direction. The code in the `if` statement sets the texture coordinates in the `VertexArray` using the coordinates returned from the `getCurrentFrame` function placed in `m_SectionToDraw` and then copied into `uPos`, `vPos`, `texWidth`, and `texHeight`.

Add this next code to the `draw` function:

```
    else if (m_PlayerUpdate->m_LeftIsHeldDown &&
            !m_PlayerUpdate->m_InJump &&
            !m_PlayerUpdate->m_BoostIsHeldDown)
        {
            canvas[m_VertexStartIndex].texCoords.x
                = uPos;
            canvas[m_VertexStartIndex].texCoords.y
                = vPos;
            canvas[m_VertexStartIndex + 1].texCoords.x
                = uPos - texWidth;
            canvas[m_VertexStartIndex + 1].texCoords.y
                = vPos;
            canvas[m_VertexStartIndex + 2].texCoords.x
                = uPos - texWidth;
```

```
            canvas[m_VertexStartIndex + 2].texCoords.y
                = vPos + texHeight;
            canvas[m_VertexStartIndex + 3].texCoords.x
                = uPos;
            canvas[m_VertexStartIndex + 3].texCoords.y
                = vPos + texHeight;
        }
```

In the preceding section of the `draw` function, the `if` statement executes when the player is holding left down, not jumping, and not boosting. This is the exact opposite of the previous `if` statement and covers when the player is running left. At first glance, the code might look the same but there is one small change to the way the width of the horizontal coordinates of the textures are handled. The second and third vertices have their `x` coordinates calculated like this:

```
= uPos - texWidth;
```

The first and third coordinates are calculated like this:

```
= uPos;
```

This has the effect of using the pixels on the right of the image in the texture on the left-hand side of the quad and moving right to left through the texture's pixels and left to right on the quad. Basically, this has horizontally flipped the image. This is just what we need when the player is facing left.

Add the following to the `draw` function:

```
    else if (m_PlayerUpdate->m_RightIsHeldDown &&
            m_PlayerUpdate->m_BoostIsHeldDown)
        {
            canvas[m_VertexStartIndex].texCoords.x =
```

```
            BOOST_TEX_LEFT;
        canvas[m_VertexStartIndex].texCoords.y =
            BOOST_TEX_TOP;
        canvas[m_VertexStartIndex + 1].texCoords.x =
            BOOST_TEX_LEFT + BOOST_TEX_WIDTH;
        canvas[m_VertexStartIndex + 1].texCoords.y =
            BOOST_TEX_TOP;
        canvas[m_VertexStartIndex + 2].texCoords.x =
            BOOST_TEX_LEFT + BOOST_TEX_WIDTH;
        canvas[m_VertexStartIndex + 2].texCoords.y =
            BOOST_TEX_TOP + BOOST_TEX_HEIGHT;
        canvas[m_VertexStartIndex + 3].texCoords.x =
            BOOST_TEX_LEFT;
        canvas[m_VertexStartIndex + 3].texCoords.y =
            BOOST_TEX_TOP + BOOST_TEX_HEIGHT;
    }
```

In the preceding section, the `if` statement executes when the player is holding right and boosting. The texture coordinates in the `if` statement are set using the integer constants that represent the boosting graphic from the texture atlas. These are `BOOST_TEX_LEFT`, `BOOST_TEX_TOP`, `BOOST_TEX_WIDTH`, and `BOOST_TEX_HEIGHT`.

Add the following to the `draw` function:

```
else if (m_PlayerUpdate->m_LeftIsHeldDown &&
        m_PlayerUpdate->m_BoostIsHeldDown)
    {
        canvas[m_VertexStartIndex].texCoords.x =
            BOOST_TEX_LEFT + BOOST_TEX_WIDTH;
        canvas[m_VertexStartIndex].texCoords.y = 0;
        canvas[m_VertexStartIndex + 1].texCoords.x =
            BOOST_TEX_LEFT;
        canvas[m_VertexStartIndex + 1].texCoords.y = 0;
        canvas[m_VertexStartIndex + 2].texCoords.x =
            BOOST_TEX_LEFT;
        canvas[m_VertexStartIndex + 2].texCoords.y = 100;
        canvas[m_VertexStartIndex + 3].texCoords.x =
            BOOST_TEX_LEFT + BOOST_TEX_WIDTH;
```

```
            canvas[m_VertexStartIndex + 3].texCoords.y = 100;
        }
```

In the preceding `if` statement, the code executes when the player is boosting to the left. Again, we use the constants that represent the boosting image and, as with when the player was running to the left, we flip the horizontal coordinates and read the pixels right to left to make the player graphic face left when drawn to the screen.

Add the following to the `draw` function:

```
    else if (m_PlayerUpdate->m_BoostIsHeldDown)
       {
           canvas[m_VertexStartIndex].texCoords.x
               = BOOST_TEX_LEFT;
           canvas[m_VertexStartIndex].texCoords.y
               = BOOST_TEX_TOP;
           canvas[m_VertexStartIndex + 1].texCoords.x
               = BOOST_TEX_LEFT + BOOST_TEX_WIDTH;
           canvas[m_VertexStartIndex + 1].texCoords.y
               = BOOST_TEX_TOP;
           canvas[m_VertexStartIndex + 2].texCoords.x
               = BOOST_TEX_LEFT + BOOST_TEX_WIDTH;
           canvas[m_VertexStartIndex + 2].texCoords.y
               = BOOST_TEX_TOP + BOOST_TEX_HEIGHT;
           canvas[m_VertexStartIndex + 3].texCoords.x
               = BOOST_TEX_LEFT;
           canvas[m_VertexStartIndex + 3].texCoords.y
               = BOOST_TEX_TOP + BOOST_TEX_HEIGHT    }
```

In the preceding section, the `else if` statement executes when only the boost button is held. The same constants are used when boosting and holding right.

Add the following to the `draw` function:

```
else
   {
       if (m_LastFacingRight)
       {
           canvas[m_VertexStartIndex].texCoords.x =
               m_StandingStillSectionToDraw->left;
           canvas[m_VertexStartIndex].texCoords.y =
               m_StandingStillSectionToDraw->top;
           canvas[m_VertexStartIndex + 1].texCoords.x =
               m_StandingStillSectionToDraw->left + texWidth;
           canvas[m_VertexStartIndex + 1].texCoords.y =
               m_StandingStillSectionToDraw->top;
           canvas[m_VertexStartIndex + 2].texCoords.x =
               m_StandingStillSectionToDraw->left + texWidth;
           canvas[m_VertexStartIndex + 2].texCoords.y =
               m_StandingStillSectionToDraw->top + texHeight;
           canvas[m_VertexStartIndex + 3].texCoords.x =
               m_StandingStillSectionToDraw->left;
           canvas[m_VertexStartIndex + 3].texCoords.y =
               m_StandingStillSectionToDraw->top + texHeight;
       }
       else
       {
           canvas[m_VertexStartIndex].texCoords.x =
               m_StandingStillSectionToDraw->left + texWidth;
           canvas[m_VertexStartIndex].texCoords.y =
               m_StandingStillSectionToDraw->top;
           canvas[m_VertexStartIndex + 1].texCoords.x =
               m_StandingStillSectionToDraw->left;
           canvas[m_VertexStartIndex + 1].texCoords.y =
               m_StandingStillSectionToDraw->top;
           canvas[m_VertexStartIndex + 2].texCoords.x =
               m_StandingStillSectionToDraw->left;
           canvas[m_VertexStartIndex + 2].texCoords.y =
               m_StandingStillSectionToDraw->top + texHeight;
           canvas[m_VertexStartIndex + 3].texCoords.x =
               m_StandingStillSectionToDraw->left + texWidth;
           canvas[m_VertexStartIndex + 3].texCoords.y =
               m_StandingStillSectionToDraw->top + texHeight;
       }
```

```
        }
    }
```

In the preceding and final part of the `draw` function, there is a final `else` clause to all the other `if` and `else-if` statements. It is the last possibility that executes when nothing else does. It handles the case when the player is standing still. Within the `else` block is an `if` statement that executes when `m_LastFacingRight` is `true` and an `else` statement that executes when `m_LastFacingRight` is `false`. In both cases, the coordinates that were saved into `m_StandingStillSectionToDraw` are used to set the texture coordinates. In the `else` statement, however, the horizontal coordinates are flipped so the player character faces left.

Now, we can enjoy the fruits of our work and run the game.

Running the game

Temporarily change one line of code in the `LevelUpdate.h` file as shown here:

```
bool m_IsPaused = false;
```

Changing `m_isPaused` to `false` will let the platforms spawn.

Now, run the code.

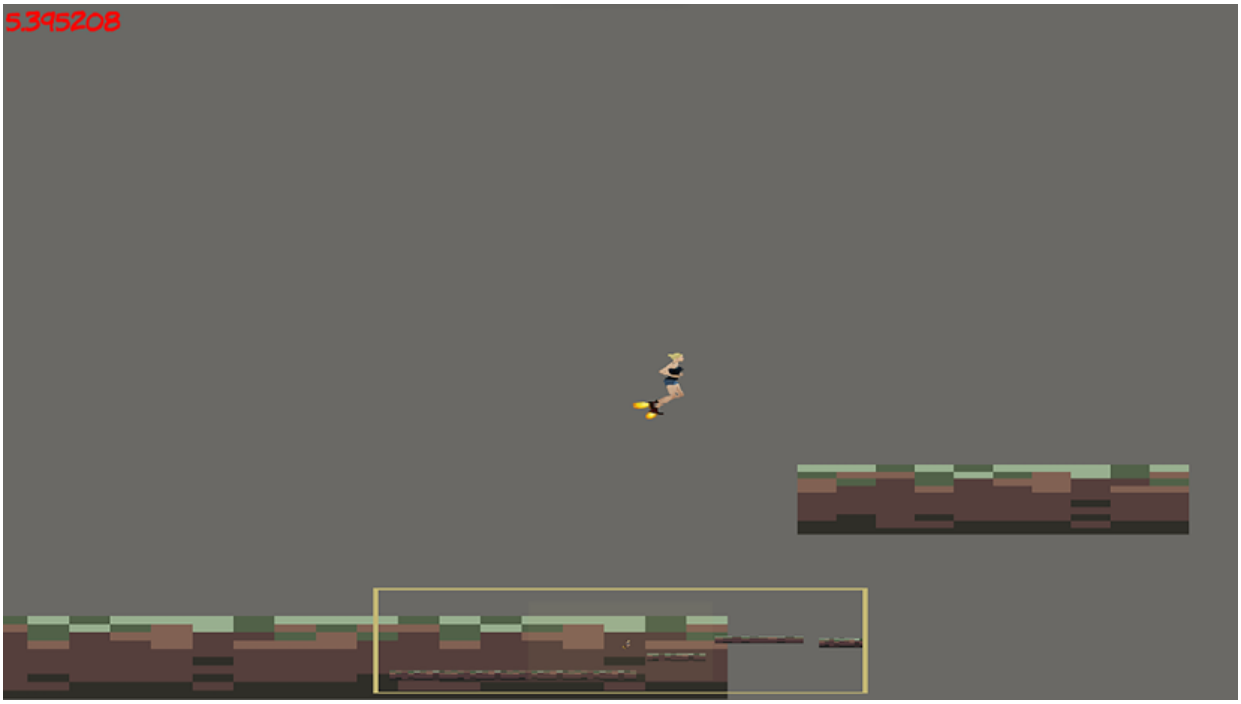


Figure 18.6: See animations

You can now run, boost, and jump until your heart is content. Be sure to test running to the left and you can see that it still looks decent because the animations are reversed.

Change `m_IsPaused` back to `true` because we will soon code a menu that handles this.

Summary

In this chapter, we coded the platforms. As expected, two classes were needed: one derived from `Update` and one from `Graphics`. We have added controls to the player, coded an `Animator` class, and put it to work in the `PlayerGraphics` class to make the player run smoothly left and right. In the next chapter, we will first build a menu to control pausing, resuming, and quitting, and then we will make it rain on the player.

19

Building the Menu and Making It Rain

In this chapter, we will implement two significant features. One of them is a menu screen to keep the player informed of their options for starting, pausing, restarting, and quitting the game. The other job will be to create a simple rain effect. You could argue that the rain effect isn't necessary, or even that it doesn't fit the game, but it is easy, fun, and a good trick to learn. What you should expect by now, and yet is still perhaps the most interesting aspect of this chapter, is how we will achieve both these objectives by coding classes derived from `Graphics` and `Update`, composing them in `GameObject` instances, and they will just work alongside all our other game entities.

This is what's coming up in this chapter:

- Building an interactive menu
- Coding the `MenuUpdate` class
- Coding the `MenuGraphics` class
- Building a menu in the factory
- Making it rain
- Coding the `RainGraphics` class
- Making it rain in the factory

The code in the `Run5` folder shows the completed state at the end of this chapter.

Building an interactive menu

To get started, let's see what the menu will look like to the player in its two possible states.

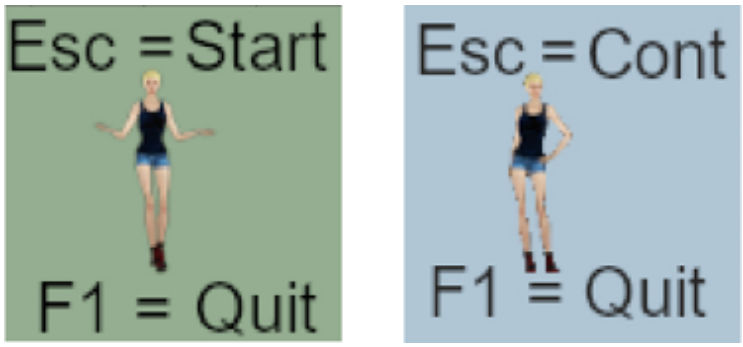


Figure 19.1: Two menu states

We can see the two possibilities in the previous image. On the left, the player is informed that they can press *Esc* to start or *F1* to quit, and on the right, the player can see that they can press *Esc* to continue or *F1* to quit. The reason for the subtle difference is that while the game is playing, they will also be able to pause by pressing *Esc*. When on either of the menu screens, *F1* will always quit, but while the game is being played, *F1* has no effect.

Coding the MenuUpdate class

Now, we will create a new class that will control our in-game menu. Create a new class called `MenuUpdate` derived from `Update` and a new class called `MenuGraphics` derived from `Graphics`.

Now, we can start coding. Add the following code to `MenuUpdate.h`:

```
#pragma once
#include "Update.h"
#include "InputReceiver.h"
#include <SFML/Graphics.hpp>
using namespace sf;
using namespace std;
class MenuUpdate :
    public Update
{
private:
    FloatRect m_Position;
    InputReceiver m_InputReceiver;
    FloatRect* m_PlayerPosition = nullptr;
    bool m_IsVisible = false;
    bool* m_IsPaused;
    bool m_GameOver;
    RenderWindow* m_Window;
public:
    MenuUpdate(RenderWindow* window);
    void handleInput();
    FloatRect* getPositionPointer();
    bool* getGameOverPointer();
    InputReceiver* getInputReceiver();

    //From Update : Component
void update(float fps) override;
    void assemble(
        shared_ptr<LevelUpdate> levelUpdate,
        shared_ptr<PlayerUpdate> playerUpdate)
 override;
};
```

What follows is an introduction to all the member variables and functions in the preceding code:

- The `FloatRect` variable called `m_Position` will hold the horizontal and vertical positions and size.
- The `InputReceiver` instance called `m_InputReceiver` will play the exact same role as the variable with the same name in `PlayerUpdate` and

`CameraUpdate` with the exception that it will respond to the *Esc* and *F1* keys as we would expect, having just discussed the menu.

- The `FloatRect` pointer `m_PlayerPosition` will track the player's position and then the menu can position itself relative to the player when it needs to be seen.
- The Boolean `m_IsVisible` variable lets the menu know when to show itself and when to hide itself.
- The Boolean pointer `m_IsPaused` will hold the address of the variable in the `LevelUpdate` class, which defines whether the game is paused. Then, in conjunction with `m_IsVisible` and `m_GameOver`, the menu will know when to show itself and which image to use. The Boolean `m_GameOver` works in conjunction with the two preceding variables.
- The `RenderWindow` pointer `m_Window` will give the menu the power to close the app window and end the app execution.
- The `MenuUpdate` constructor will be used to prepare the `MenuUpdate` class to do its job. It will handle details that the `assemble` function cannot.
- The `handleInput` function is called once each frame from the `update` function to handle the operating system events sent from the `InputDispatcher` in the main game loop.
- The `getPositionPointer` function returns a `FloatRect` pointer to the position and scale of the menu.
- The `getGameOverPointer` function returns the address of a Boolean that indicates when the game has ended because the player lost.
- The `getInputReceiver` function returns the address of the `InputReceiver` instance to the `InputDispatcher`.
- The overridden `update` function executes each iteration of the game loop. Soon, we will see the code that we put in it.

- The overridden `assemble` function, as we have come to expect, returns `void` and has the following parameters: `shared_ptr<LevelUpdate> levelUpdate` and `shared_ptr<PlayerUpdate> playerUpdate`. The function will be coded shortly to show the specific but similar way that this class uses it.

We can now move on to the implementation of the `MenuUpdate` class. We will add the code for `MenuUpdate.cpp` in four `main` sections. To make these sections work, we will first need to add the following `include` directives:

```
#include "MenuUpdate.h"
#include "LevelUpdate.h"
#include "PlayerUpdate.h"
#include "SoundEngine.h"
```

Add the first `main` section for `MenuUpdate.cpp`:

```
MenuUpdate::MenuUpdate(RenderWindow* window)
{
    m_Window = window;
}
FloatRect* MenuUpdate::getPositionPointer()
{
    return &m_Position;
}
bool* MenuUpdate::getGameOverPointer()
{
    return &m_GameOver;
}
InputReceiver* MenuUpdate::getInputReceiver()
{
    return &m_InputReceiver;
}
```

In the first section of the `MenuUpdate.cpp` code above, we have the constructor where the pointer to the `RenderWindow` is initialized. We will use the pointer to a `RenderWindow` when the player presses *F1* to shut down the game. The `getPositionPointer` function returns a pointer to `m_Position`. The other class that cares about the position of the menu is, of course, the `MenuGraphics` class, which will need to draw the menu.

The `getGameOverPointer` function returns the address of the `m_GameOver` Boolean. The `getInputReceiver` function is used (as with `PlayerUpdate` and `CameraUpdate`) to get a pointer to the `InputReceiver` instance. Again, this is used by the `InputDispatcher` so it knows where to send all the operating system events in each frame.

Next, add the `assemble` function to `MenuUpdate.cpp`:

```
void MenuUpdate::assemble(
    shared_ptr<LevelUpdate> levelUpdate,
    shared_ptr<PlayerUpdate> playerUpdate)
{
    m_PlayerPosition =
        playerUpdate->getPositionPointer();
    m_IsPaused =
        levelUpdate->getIsPausedPointer();
    m_Position.height = 75;
    m_Position.width = 75;
    SoundEngine::startMusic();
    SoundEngine::pauseMusic();
}
```

In the second section of the `MenuUpdate.cpp` code above, the `assemble` function prepares the class to be used. First, the address of the player's position is initialized to `m_PlayerPosition`, and the address of the Boolean indicating the paused state from the `LevelUpdate` instance is copied to `m_IsPaused`. Second, the width and height of the menu are defined by the

magic number, which is `75` in this case. Finally, the music starts and is then immediately paused. The music is now ready to be resumed and paused whenever the player resumes or pauses the game.

Now, add the third section for `MenuUpdate.cpp`. The `handleInput` function will be shown next:

```
void MenuUpdate::handleInput()
{
    for (const Event& event :
        m_InputReceiver.getEvents())
    {
        if (event.type ==
            Event::KeyPressed)
        {
            if (event.key.code ==
                Keyboard::F1 && m_IsVisible)
            {
                if (SoundEngine::mMusicIsPlaying)
                {
                    SoundEngine::stopMusic();
                }
                m_Window->close();
            }
        }
        if (event.type == Event::KeyReleased)
        {
            if (event.key.code ==
                Keyboard::Escape)
            {
                m_IsVisible = !m_IsVisible;
                *m_IsPaused = !*m_IsPaused;
                if (m_GameOver)
                {
                    m_GameOver = false;
                }
                if (!*m_IsPaused)
                {
                    SoundEngine::resumeMusic();
                    SoundEngine::playClick();
```

```
                }
                if (*m_IsPaused)
                {
                    SoundEngine::pauseMusic();
                    SoundEngine::playClick();
                }
            }
        }
    }
    m_InputReceiver.clearEvents();
}
```

In the third section, the `handleInputFunction` should look familiar to all the event loops we have created in the previous projects and this project. The `for` loop, which loops through all the input events for the current frame of the game, wraps two `if` statements.

The first `if` statement tests for the combination of *F1* being pressed and the menu being visible. If the music is playing, it is stopped and then the `RenderWindow` pointer is used to close the window and shut the game down.

The second `if` statement and the further nested `if` statement test for the release of the *Esc* key. First, `m_IsVisible` is switched to its opposite value. If it is `true`, it becomes `false`, and if it is `false`, it becomes `true`. This is just what we need. Every time the player taps the *Esc* key, the game state will flip between paused and playing. The exact same flip is done with `m_IsVisible` to show and hide the menu.

At this point, the Boolean states have been set correctly and the code takes the required action based on the current state. If the game is over (`m_GameOver` is `true`), `m_GameOver` is set to `false`. If the game is not paused, then the music is resumed and a click sound is played, and finally, if the game is paused, the music is paused and a click sound is played.

Outside of the event `for` loop, all the events are cleared from the `m_InputReceiver` instance ready for the next iteration of the main game loop by calling `clearEvents`.

Finally, for `MenuUpdate.cpp`, add the following code for the `update` function:

```
void MenuUpdate::update(float fps)
{
    handleInput();
    if (*m_IsPaused && !m_IsVisible)// Game over 1
    {
        m_IsVisible = true;
        m_GameOver = true;
    }
    if (m_IsVisible)
    {
        // Follow the player
        m_Position.left =
            m_PlayerPosition->getPosition()-x -
            m_Position.width / 2;
        m_Position.top =
            m_PlayerPosition->getPosition()-y -
            m_Position.height / 2;
    }
    else
    {
        m_Position.left = -999;
        m_Position.top = -999;
    }
}
```

In the preceding final section, the `update` function of the `MenuUpdate` class starts by calling the `handleInput` function that we coded previously. The first `if` statement executes when the game is paused, and the menu is not visible. The code in the `if` statement sets `paused` to `true` and `m_GameOver` to `true`.

The second `if` statement in the `update` function executes when the menu is visible. When the menu is visible, it makes sense that we need to make sure the menu is actually visible. To this end, `m_Position.left` and `m_Position.top` are initialized to the left and top positions of the player, respectively, minus the width and height of the player. This has the effect of positioning the menu in the center of the screen, over the player.

In the final `else` clause, which executes when the game is not paused, we initialize `m_Position.left` and `m_Position.top` to `-999`, which has the effect of hiding the menu.

Now, we can move on to the `MenuGraphics` class and see how `MenuUpdate` and `MenuGraphics` complement each other.

Coding the MenuGraphics class

To get started, add the following code to the `MenuGraphics.h` file:

```
#pragma once
#include "Graphics.h"
#include "SFML/Graphics.hpp"
class MenuGraphics :
    public Graphics
{
private:
    FloatRect* m_MenuPosition = nullptr;
    int m_VertexStartIndex;
    bool* m_GameOver;
    bool m_CurrentStatus = false;
    int uPos;
    int vPos;
    int texWidth;
    int texHeight;
public:
    // From Graphics : Component
void draw(VertexArray& canvas) override;
```

```
        void assemble(VertexArray& canvas,
            shared_ptr<Update> genericUpdate,
            IntRect texCoords) override;
    };
```

In the preceding code, the `FloatRect` pointer `m_MenuPosition` is initialized to `nullptr`. This variable will soon be initialized to track the `m_Position` `FloatRect` instance from the `MenuUpdate` class.

The integer `m_VertexStartIndex` variable will be initialized to remember the starting index of the quad of vertices that represent the menu in the `VertexArray` that is drawn for each frame of the game.

The Boolean pointer `m_GameOver` will be initialized to track the `m_GameOver` variable from the `MenuUpdate` class.

The Boolean `m_CurrentStatus` will be used to make decisions and remember them by initializing to `m_GameOver` and then testing for changes. It will make more sense when we see the code in `MenuGraphics.cpp`.

The integer `uPos` will hold the horizontal texture coordinate, `vPos` will hold the vertical texture coordinate, `texWidth` the texture width, and `texHeight` the texture height.

The first public function is the overridden `draw` function, and we are more than familiar with its signature; we will see how to code it soon.

The `assemble` function has the usual suspects as parameters and we have seen it multiple times. We will code it very soon to see how we assemble the `MenuGraphics` class.

For the `MenuGraphics.cpp` file, we will code this in two sections: the `assemble` function and the `draw` function.

Add the following `assemble` function to `MenuGraphics.cpp`:

```
#include "MenuGraphics.h"
#include "MenuUpdate.h"
void MenuGraphics::assemble(
    VertexArray& canvas,
    shared_ptr<Update> genericUpdate,
    IntRect texCoords)
{
    m_MenuPosition = static_pointer_cast<MenuUpdate>(
        genericUpdate)->getPositionPointer();
    m_GameOver = static_pointer_cast<MenuUpdate>(
        genericUpdate)->getGameOverPointer();
    m_CurrentStatus = *m_GameOver;
    m_VertexStartIndex = canvas.getVertexCount();
    canvas.resize(canvas.getVertexCount() + 4);
    // Remember the UV coordinates
// because we manipulate them later
    uPos = texCoords.left;
    vPos = texCoords.top;
    texWidth = texCoords.width;
    texHeight = texCoords.height;
    canvas[m_VertexStartIndex].texCoords.x = uPos;
    canvas[m_VertexStartIndex].texCoords.y =
        vPos + texHeight;
    canvas[m_VertexStartIndex + 1].texCoords.x =
        uPos + texWidth;
    canvas[m_VertexStartIndex + 1].texCoords.y =
        vPos + texHeight;
    canvas[m_VertexStartIndex + 2].texCoords.x =
        uPos + texWidth;
    canvas[m_VertexStartIndex + 2].texCoords.y =
        vPos + texHeight + texHeight;
    canvas[m_VertexStartIndex + 3].texCoords.x =
        uPos;
    canvas[m_VertexStartIndex + 3].texCoords.y =
        vPos + texHeight + texHeight;
}
```

The preceding `assemble` function code starts with this line:

```
        m_MenuPosition = static_pointer_cast<MenuUpdate>(
            genericUpdate)->getPositionPointer();
```

The `static_pointer_cast` function is casting a `genericUpdate` instance, which is currently of type `Update`, into a specific `MenuUpdate` `shared_ptr` instance. In the same line of code after the conversion, the `getPositionPointer` function is called on the `MenuUpdate` instance. The returned result of this function call is stored in `m_MenuPosition`.

The next line of code uses the same casting technique but, instead, calls the `getGameOverPointer` function and stores the result in `m_GameOver`.

Next, `m_GameOver` is dereferenced, and the integer value is stored in `m_CurrentStatus`.

Following on, the `m_StartIndex` variable is initialized by getting the current size of `VertexArray`, and the `VertexArray` size is then increased to fit another quad by calling `canvas.resize`.

Next, we remember the texture coordinates by initializing `uPos`, `vPos`, `texWidth`, and `texHeight` from the passed-in texture coordinates. Remember, these are stored and will soon be passed in from the `Factory` class.

The next eight lines of code initialize the texture coordinates directly into `VertexArray`. The reason we need to store the original texture coordinates separate from the `VertexArray` is that we will soon add code in the `update` function that manipulates the texture coordinates to show the different versions of our menu for when the game is paused (as opposed to the game being over).

Finally, for `MenuGraphics.cpp` and the `MenuGraphics` class, add the following `draw` function to `MenuGraphics.cpp`:

```
void MenuGraphics::draw(VertexArray& canvas)
{
    if (*m_GameOver && !m_CurrentStatus)
    // current status has just switched to game over
    {
        // Each v coordinate is doubled to
// reference the texture below
        m_CurrentStatus = *m_GameOver;
        canvas[m_VertexStartIndex].texCoords.x =
            uPos;
        canvas[m_VertexStartIndex].texCoords.y =
            vPos + texHeight;
        canvas[m_VertexStartIndex + 1].texCoords.x =
            uPos + texWidth;
        canvas[m_VertexStartIndex + 1].texCoords.y =
            vPos + texHeight;
        canvas[m_VertexStartIndex + 2].texCoords.x =
            uPos + texWidth;
        canvas[m_VertexStartIndex + 2].texCoords.y =
            vPos + texHeight + texHeight;
        canvas[m_VertexStartIndex + 3].texCoords.x =
            uPos;
        canvas[m_VertexStartIndex + 3].texCoords.y =
            vPos + texHeight + texHeight;
    }
    else if (!*m_GameOver && m_CurrentStatus)
    {
        m_CurrentStatus = *m_GameOver;
        canvas[m_VertexStartIndex].texCoords.x =
            uPos;
        canvas[m_VertexStartIndex].texCoords.y =
            vPos;
        canvas[m_VertexStartIndex + 1].texCoords.x =
            uPos + texWidth;
        canvas[m_VertexStartIndex + 1].texCoords.y =
            vPos;
        canvas[m_VertexStartIndex + 2].texCoords.x =
            uPos + texWidth;
```

```
            canvas[m_VertexStartIndex + 2].texCoords.y =
                vPos + texHeight;
            canvas[m_VertexStartIndex + 3].texCoords.x =
                uPos;
            canvas[m_VertexStartIndex + 3].texCoords.y =
                vPos + texHeight;
        }
        const Vector2f& position =
            m_MenuPosition->getPosition();
        canvas[m_VertexStartIndex].position =
            position;
        canvas[m_VertexStartIndex + 1].position =
            position + Vector2f(m_MenuPosition->getSize().x, 0);
        canvas[m_VertexStartIndex + 2].position =
            position + m_MenuPosition->getSize();
        canvas[m_VertexStartIndex + 3].position =
            position + Vector2f(0, m_MenuPosition->getSize().y);
    }
```

In the preceding `draw` function, there is an `if` branch and an `else if` branch to the code. The `if` branch executes when `m_GameOver` is `true` and `m_CurrentStatus` is `false`. The `else if` branch executes when `m_GameOver` is `false` and `m_CurrentStatus` is `false`.

First, let's look at what happens in the `if` branch. In the `if` branch, `m_CurrentStatus` is set to the dereferenced value of `m_GameOver` and then all of the texture coordinates in the `VertexArray` are set. The way they are set is with the same values we used in the `assemble` function. These values map to the lower version of the menu in the texture atlas.

Next, let's look at what happens in the `else if` branch. In the `else if` branch, the `m_CurrentStatus` is synchronized with `m_GameOver` again, and all of the texture coordinates in the `VertexArray` are set. However, look at the code for all of the vertical coordinates. They all lack an additional `+texHeight`. This means the coordinates now map to the upper version of the

menu graphic in the texture atlas. The texture coordinates will be flipped every time the player loses a game and every time the player pauses a game after restarting. Therefore, the texture coordinates will always map to either the paused menu or the game over menu.

Of course, we haven't positioned the vertices yet. We must do so because these positions will be regularly changing in the `MenuUpdate` class as the menu is shown and hidden. What happens after the `if-else-if` structure we have just discussed is that the vertex positions in the `VertexArray` are positioned using the values in `m_MenuPosition`, which points to the `FloatRect` in `MenuUpdate`. To make the code less verbose, we first initialize a constant `Vector2f` by calling `m_MenuPosition->getPosition`.

Building a menu in the factory

Now we can instantiate a working menu by composing a `GameObject` instance with our two new classes. Add the `include` directives for our two menu-related classes:

```
#include "MenuUpdate.h"
#include "MenuGraphics.h"
```

Next, add the following code just before the closing curly brace of the `loadLevel` function in the `Factory.cpp` file:

```
// Menu
GameObject menu;
shared_ptr<MenuUpdate> menuUpdate =
  make_shared<MenuUpdate>(m_Window);
menuUpdate->assemble(levelUpdate,
  playerUpdate);
inputDispatcher.registerNewInputReceiver(
```

```
    menuUpdate->getInputReceiver());
  menu.addComponent(menuUpdate);
  shared_ptr<MenuGraphics>menuGraphics =
    make_shared<MenuGraphics>();
  menuGraphics->assemble(canvas, menuUpdate,
    IntRect(TOP_MENU_TEX_LEFT, TOP_MENU_TEX_TOP,
      TOP_MENU_TEX_WIDTH, TOP_MENU_TEX_HEIGHT));
  menu.addComponent(menuGraphics);
  gameObjects.push_back(menu);
  // End menu
```

The preceding code should feel familiar. In the following order, we do this:

1. Create a new `GameObject` instance called `menu`.
2. Create a shared pointer to a `MenuUpdate` instance called `menuUpdate`.
3. Call the `assemble` function on `menuUpdate` passing in the `levelUpdate` and `playerUpdate` shared pointers.
4. Prepare the `menu` to receive updates by calling `registerNewInputReceiver` on `inputDispatcher` and passing in the returned result from `menuUpdate->getInputReceiver`.
5. Add `menuUpdate` to the `menu` `GameObject` instance.
6. Create a shared pointer of the `MenuGraphics` type.
7. Call the `menuGraphics` `assemble` function and pass in the `VertexArray`, the `LevelUpdate` instance, and all the required texture coordinates.
8. Add/compose the `MenuGraphics` instance to the `GameObject` instance.
9. Finally, add the `GameObject` instance, which represents our menu, to the `gameObjects` vector.

That's it. Our menu is done.

Running the game

Now you can run the game. You will see the menu shown next if you press *Esc*:

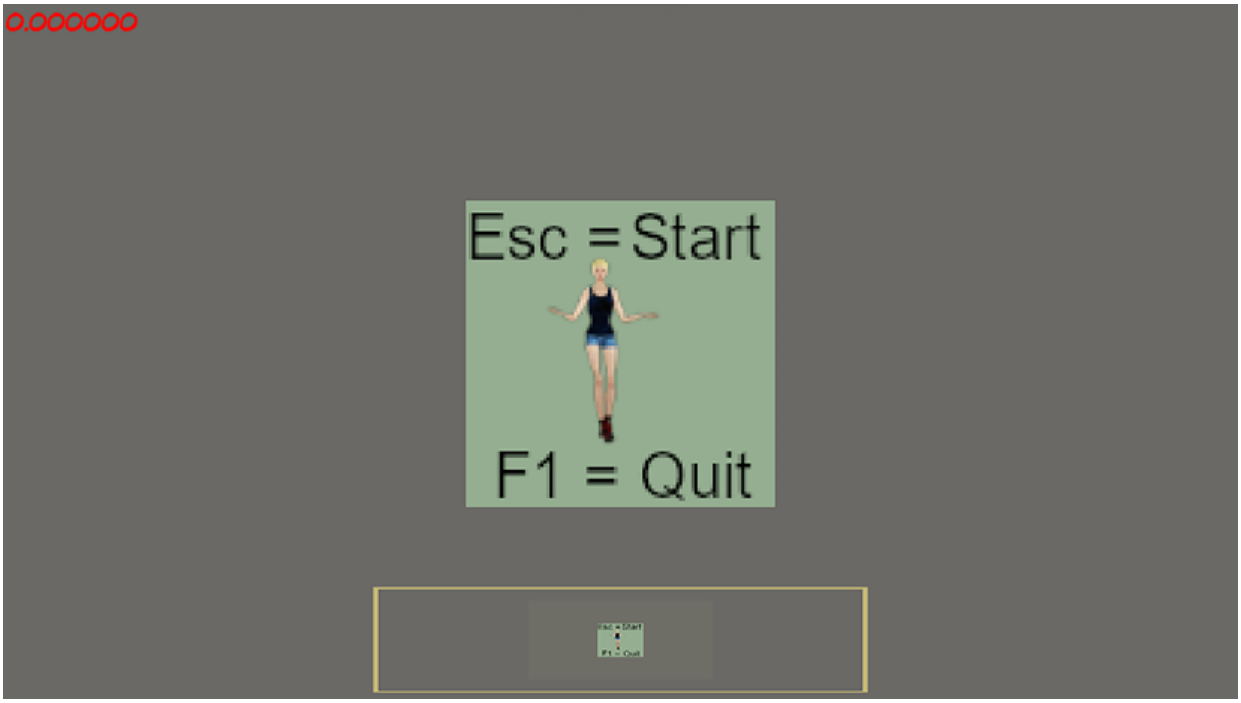


Figure 19.2: Menu

You will be able to press *F1* to quit, as suggested on the menu. However, if you try and press *Esc* to start, it will also quit the game. This is not the behavior we want. We need to make two quick changes.

We need to delete the temporary code we added to the `InputDispatcher.cpp` class back when we first started the project. Locate the following lines of code and delete them:

```
if (event.type == Event::KeyPressed &&
    event.key.code == Keyboard::Escape)
{
    m_Window->close();
}
```

You can now run the game, start and pause with *Esc*, and quit with *F1* when the menu is visible. The `InputDispatcher` class no longer handles any events; it just dispatches them to the `InputReceiver` instances in the menu, the player, and the mini-map camera.

We also need to stop the game from starting automatically. We do this in the `LevelUpdate.h` class.

Find this next line of code:

```
bool m_IsPaused = false;
```

Now change it to this:

```
bool m_IsPaused = true;
```

Now our pausing, starting, and quitting works as expected.

Making it rain

We only need a graphics component. This is fine, just as it was OK to only have an `update` component for the level logic. While the `RainGraphics` state will change, it is not dependent in any way on the timing of the main game loop or the player's input. All the changes in state are controlled by the `Animator` class instance inside the `RainGraphics` class, which has its own internal clock. We will spawn multiple instances of the `RainGraphics` class because each instance will cover a small section of the screen. Each `RainGraphics` instance will position itself relative to the player and follow the player through the world, giving the impression that it is raining everywhere.

Coding the RainGraphics class

The graphics in the texture atlas look like the following image:

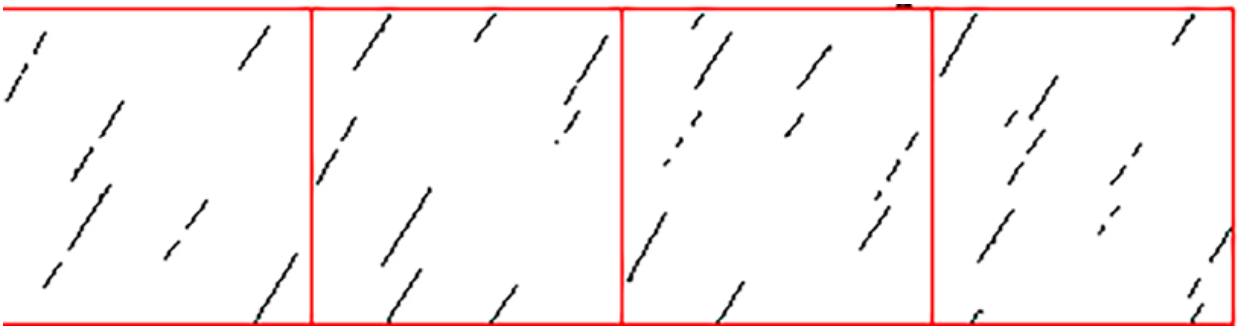


Figure 19.3: Rain From Atlas

I have drawn frames in red around each frame of animation and changed the background from transparent to white. Each frame of animation is 100 pixels by 100 pixels. Therefore, overall, the rain sprite sheet is 400 by 100 pixels. All the frames are lined up neatly ready to be looped through by our `Animator` class, which we also used to animate the player.

Create a new class called `RainGraphics` and, in the `RainGraphics.h` file, add this code:

```
#pragma once
#include "Graphics.h"
class Animator;
class RainGraphics :
    public Graphics
{
private:
    FloatRect* m_PlayerPosition;

    int m_VertexStartIndex;
    Vector2f m_Scale;
    float m_HorizontalOffset;
    float m_VerticalOffset;
```

```
    Animator* m_Animator;
    IntRect* m_SectionToDraw;
public:
    RainGraphics(FloatRect* playerPosition,
        float horizontalOffset,
        float verticalOffset,
        int rainCoveragePerObject);

    // From Graphics : Component
void draw(VertexArray& canvas) override;
    void assemble(VertexArray& canvas,
        shared_ptr<Update> genericUpdate,
        IntRect texCoords) override;
};
```

In the preceding `RainGraphics.h` code, the `FloatRect` pointer `m_PlayerPosition` will track the player's position so the rain can follow the player around like an unlucky character being followed by a gray cloud in a cartoon.

The `m_VertexStartIndex` remembers the starting index of the vertices in the `VertexArray`.

The `m_Scale` variable remembers the size of a frame of animation. The float `variable` `m_HorizontalOffset` is the starting horizontal value for the graphics in the texture atlas, and the `float` variable `m_VerticalOffset` is the vertical equivalent.

The `m_Animator` instance is our `Animator` and the `IntRect` pointer `m_SectionToDraw` will hold the texture coordinates of the current frame of animation.

The `RainGraphics` constructor receives and initializes some of the variables we have just discussed as well as an integer called `rainCoveragePerObject`, which helps us scale each rain instance.

The `draw` and `assemble` functions are the usual overridden functions that have identical declarations as before, but their implementations will be interesting and we will discuss them soon.

Next, for the `RainGraphics.cpp` file, let's code this in two sections, first the constructor and then the `assemble` function. Add the following code to `RainGraphics.cpp`:

```
#include "RainGraphics.h"
#include "RainGraphics.h"
#include "Animator.h"
RainGraphics::RainGraphics(
    FloatRect* playerPosition,
    float horizontalOffset,
    float verticalOffset,
    int rainCoveragePerObject)
{
    m_PlayerPosition = playerPosition;
    m_HorizontalOffset = horizontalOffset;
    m_VerticalOffset = verticalOffset;
    m_Scale.x = rainCoveragePerObject;
    m_Scale.y = rainCoveragePerObject;
}
```

In the constructor, we initialize the player's position pointer, the horizontal and vertical offset for the animator, and `m_Scale`, which is an `IntRect`, using the same value for `x` and `y`. We will see this in use soon.

Next, add the `assemble` function, as shown:

```
void RainGraphics::assemble(VertexArray& canvas,
    shared_ptr<Update> genericUpdate,
    IntRect texCoords)
{
    m_Animator = new Animator(
        texCoords.left,
        texCoords.top,
```

```
            4,// Frames
            texCoords.width * 4,
            texCoords.height,
            8);// FPS
        m_VertexStartIndex = canvas.getVertexCount();
        canvas.resize(canvas.getVertexCount() + 4);
    }
```

In the `assemble` function, we initialize our `Animator` instance by calling `new` and passing in the required parameters. Notice there are four frames as expected and we have specified eight frames per second.

The start index is remembered and the `VertexArray` has four spaces added to it for the quad that represents this block of rain, just as we have done so many times before. However, remember there will be multiple instances of the `RainGraphics` class.

Finally, add the `draw` function to `RainGraphics.cpp`, as shown next:

```
void RainGraphics::draw(VertexArray& canvas)
{
    const Vector2f& position =
        m_PlayerPosition->getPosition()
        - Vector2f(m_Scale.x / 2 + m_HorizontalOffset,
            m_Scale.y / 2 + m_VerticalOffset);
    // Move the rain to keep up with the player
    canvas[m_VertexStartIndex].position = position;
    canvas[m_VertexStartIndex + 1].position =
        position + Vector2f(m_Scale.x, 0);
    canvas[m_VertexStartIndex + 2].position =
        position + m_Scale;
    canvas[m_VertexStartIndex + 3].position =
        position + Vector2f(0, m_Scale.y);
    //Cycle trough the frames
    m_SectionToDraw =
        m_Animator->getCurrentFrame(false);
    // Remember the section of the texture to draw
const int uPos = m_SectionToDraw->left;
```

```
        const int vPos = m_SectionToDraw->top;
        const int texWidth = m_SectionToDraw->width;
        const int texHeight = m_SectionToDraw->height;
        canvas[m_VertexStartIndex].texCoords.x =
            uPos;
        canvas[m_VertexStartIndex].texCoords.y =
            vPos;
        canvas[m_VertexStartIndex + 1].texCoords.x =
            uPos + texWidth;
        canvas[m_VertexStartIndex + 1].texCoords.y =
            vPos;
        canvas[m_VertexStartIndex + 2].texCoords.x =
            uPos + texWidth;
        canvas[m_VertexStartIndex + 2].texCoords.y =
            vPos + texHeight;
        canvas[m_VertexStartIndex + 3].texCoords.x =
            uPos;
        canvas[m_VertexStartIndex + 3].texCoords.y =
            vPos + texHeight;
    }
```

In the `draw` function, the first section of code uses the player's position to move the rain to keep up with the player wherever it may have moved. Notice that in the first line of code, the `m_HorizontalOffset` and `m_VerticalOffset` values are used to make sure the instance is positioned in its correct place relative to all the other `RainGraphics` instances. These offset values, you may remember, were passed into the constructor that we coded previously. As you might have come to expect, the multiple `RainGraphics` instances and their offsets will be coordinated when they are created in the `Factory` class.

Next, the `getCurrentFrame` function of the `Animator` class is called to get the current texture coordinates and the usual eight lines of code assign the appropriate `x` and `y` coordinates to the four vertices of the quad.

Making it rain in the factory

To get started, add the following `include` directive to `Factory.cpp`:

```
#include "RainGraphics.h"
```

Add the following code to create multiple `RainGraphics` instances just after the `platform` code and before the `camera` code:

```
 // Rain
int rainCoveragePerObject = 25;
  int areaToCover = 350;
  for (int h = -areaToCover / 2;
    h < areaToCover / 2;
    h += rainCoveragePerObject)
  {
    for (int v = -areaToCover / 2;
      v < areaToCover / 2;
      v += rainCoveragePerObject)
    {
      GameObject rain;
      shared_ptr<RainGraphics> rainGraphics =
        make_shared<RainGraphics>(
          playerUpdate->getPositionPointer(),
          h, v, rainCoveragePerObject);
      rainGraphics->assemble(
        canvas, nullptr,
        IntRect(RAIN_TEX_LEFT, RAIN_TEX_TOP,
          RAIN_TEX_WIDTH, RAIN_TEX_HEIGHT));
      rain.addComponent(rainGraphics);
      gameObjects.push_back(rain);
    }
  }
  //End rain
```

In the preceding code, we do the usual things: make a `GameObject` instance, make a `RainGraphics` instance, add the `RainGraphics` instance to the

`GameObject`, and add the `GameObject` to the `vector` of `GameObjects`. Why not spend a moment identifying these parts that have become familiar by now? Then, I will describe what is new.

What is new is that we first declare some extra variables to control the position and size of multiple `RainGraphics` instances, as follows:

```
int rainCoveragePerObject = 25;
int areaToCover = 350;
```

Next, note the structure of the `for` loop that iterates and creates multiple instances. Here it is again:

```
for (int h = -areaToCover / 2;
    h < areaToCover / 2;
    h += rainCoveragePerObject)
```

The condition of the `for` loop means that `h` will go from `-175` to `175` and that it will do so in increments of `25`. Inside the `RainGraphics` instances, all these values are world units, not pixels.

The inner `for` loop that initializes the `v` parameter uses the same formula as the outer `for` loop. Finally, note the call to the constructor of the `RainGraphics` class:

```
shared_ptr<RainGraphics> rainGraphics =
        make_shared<RainGraphics>(
          playerUpdate->getPositionPointer(),
          h, v, rainCoveragePerObject);
```

Overall, this has the effect of looping a 14 x 14 block (196) of `RainGraphics` instances around the player.

Running the game

Now we can see the fruits of our work and run the game.

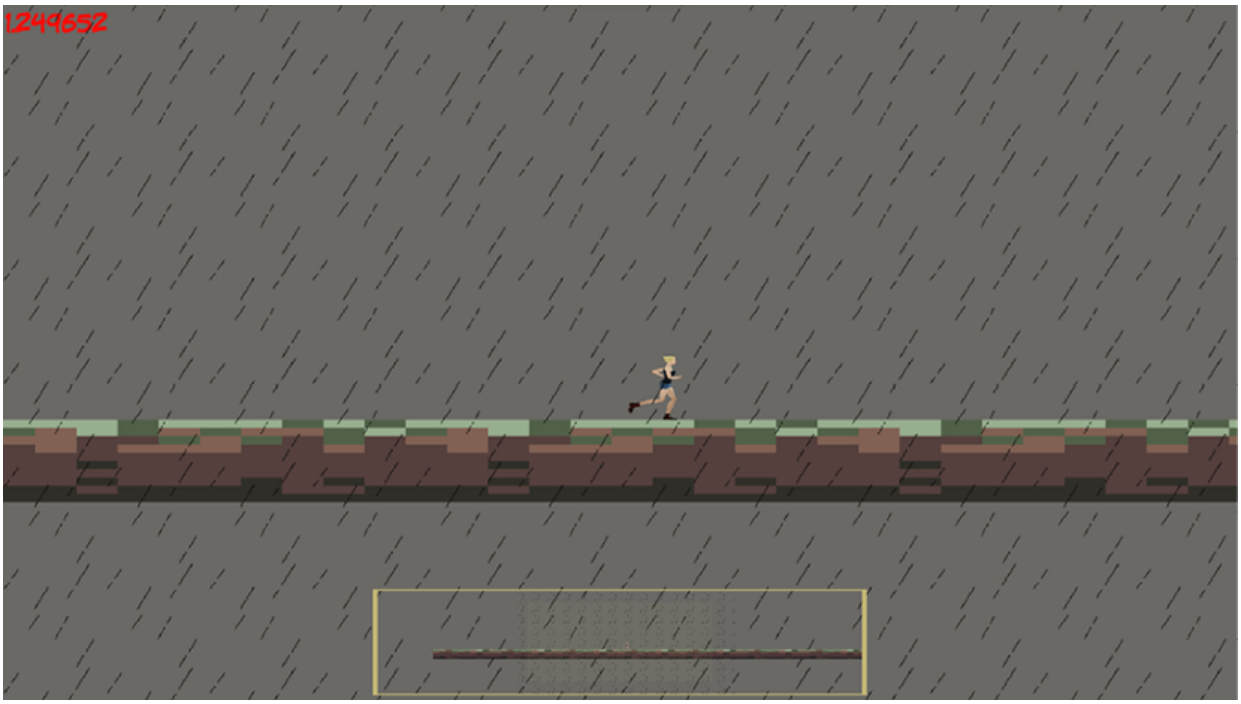


Figure 19.4: Rain

That's it for rain!

Summary

In this chapter, we have created an interactive menu with two possible appearances by coding the `MenuUpdate` class and the `MenuGraphics` class. Afterward, we created a menu in the factory in the exact same way we have been adding features to our game over the previous few chapters.

To finish, we created a new `Graphics`-derived class called `RainGraphics`, which creates a simple and effective rain effect. As usual, we wrapped this

class in a `GameObject` instance in the factory, popped it in the `gameObjects` vector, and hey presto, it just works.

In the next chapter, we will add fireballs to the game that fly in from the left or right to disrupt the player's progress.

20

Fireballs and Spatialization

In this chapter, we will be adding all the sound effects and the HUD. We did this in two of the previous projects, but we will do things a bit differently this time. We will explore the concept of sound **spatialization** and how SFML makes this complicated concept nice and easy. In addition, we will build a HUD class to encapsulate our code that draws information to the screen.

We will cover the following topics in this chapter:

- What is spatialization?
- Handling spatialization using SFML
- Upgrading the `SoundEngine` class
- Coding the fireball-related classes (derived from `Graphics` and `Update`) that make spatialized sounds
- Building some fireballs in the factory
- Running the code

The complete code for this chapter can be found in the `Run6` folder.

What is spatialization?

Spatialization is the act of making something relative to the space it is a part of, or within. In our daily lives, everything in the natural world, by default, is

spatialized. If a motorbike whizzes past from left to right, we will hear the sound grow from faint to loud, from one side to the other. As it passes by, it will become more prominent in the other ear, before fading into the distance once more. If we were to wake up one morning and the world was no longer spatialized, it would be exceptionally weird.

If we can make our video games a little bit more like the real world, our players will become more immersed. Our zombie game would have been a lot more fun if the players could have heard the zombies faintly in the distance and more loudly as they drew closer, from one direction or another.

It is probably obvious that the mathematics of spatialization is complex. How do we calculate how loud a given sound will be in a specific speaker, based on the distance and direction from the player (the hearer of the sound) to the object that is making the sound (the emitter)?

Fortunately, SFML does all the complicated processes for us. All we need to do is get familiar with a few technical terms and then we can start using SFML to spatialize our sound effects.

Emitters, attenuation, and listeners

We will need to be aware of a few pieces of information to give SFML what it needs to do its work. We will need to be aware of where the sound is coming from in our game world. This source of the sound is called an **emitter**. In a game, the emitter could be a zombie, a vehicle, or, in the case of our current project, a fire tile. We have already been keeping track of the position of the objects in our game, so giving SFML the emitter's location will be quite straightforward.

The next factor we need to be aware of is **attenuation**. Attenuation is the rate at which a wave deteriorates. You could simplify that statement and make it specific to sound and say that attenuation is how quickly the sound reduces in volume. It isn't technically accurate, but it is a good enough description for this chapter and our game.

The final factor that we need to consider is the **listener**. When SFML spatializes the sound, where is it spatializing it relative to; where are the "ears" of the game? In most games, the logical thing to do is to use the player character as the "ears" of the game.

Let's look at some hypothetical code before we see the code for real.

Handling spatialization using SFML

SFML has several functions that allow us to handle emitters, attenuation, and listeners. Let's take a look at them hypothetically, and then we will write some code to add spatialized sound to our project for real.

We can set up a sound effect that is ready to be played, as we have already done so often, like this:

```
// Declare SoundBuffer in the usual way
SoundBuffer zombieBuffer;
// Declare a Sound object as-per-usual
Sound zombieSound;
// Load the sound from a file like we have done so often
zombieBuffer.loadFromFile("sound/zombie_growl.wav");
// Associate the Sound object with the Buffer
zombieSound.setBuffer(zombieBuffer);
```

We can set the position of the emitter using the `setPosition` function shown in the following code:

```
// Set the horizontal and vertical positions of the emitter
// In this case the emitter is a zombie
// In the Zombie Arena project we could have used
// getPosition().x and getPosition().y
// These values are arbitrary
float x = 500;
float y = 500;
zombieSound.setPosition(x, y, 0.0f);
```

As suggested in the comments of the previous code, how exactly we can obtain the coordinates of the emitter will probably be dependent on the type of game. As shown in the previous code, this would be quite simple in the Zombie Arena project. We will have a few challenges to overcome when we set the position in this project.

We can set the attenuation level as follows:

```
zombieSound.setAttenuation(15);
```

The actual attenuation level can be a little ambiguous. The effect that we want the player to get might be different from the accurate scientific formula that is used to reduce the volume over distance based on attenuation. Getting the right attenuation level is usually achieved by experimenting. The higher the level of attenuation, the quicker the sound level reduces to silence.

Also, we might want to set a zone around the emitter where the volume is not attenuated at all. We might do this if the feature isn't appropriate beyond a certain range or if we have many sound sources and don't want to overdo the feature. To do so, we can use the `setMinimumDistance` function, as shown here:

```
zombieSound.setMinDistance(150);
```

With the previous line of code, attenuation would not be calculated until the listener is 150 pixels/unit away from the emitter.

Some other useful functions from the SFML library include the `setLoop` function. This function will tell SFML to keep playing the sound over and over when `true` is passed in as a parameter, like in the following code:

```
zombieSound.setLoop(true);
```

The sound will continue to play until we end it with the following code:

```
zombieSound.stop();
```

From time to time, we want to know the status of a sound (playing or stopped). We can achieve this with the `getStatus` function, as demonstrated in the following code:

```
if (zombieSound.getStatus() == Sound::Status::Stopped)
{
    // The sound is NOT playing
// Take whatever action here
}
if (zombieSound.getStatus() == Sound::Status::Playing)
{
    // The sound IS playing
// Take whatever action here
}
```

There is just one more aspect of using sound spatialization with SFML that we need to cover, the listener. Where is the listener? We can set the position of the listener with the following code:

```
// Where is the listener?
// How we get the values of x and y varies depending upon the ga
// In the Zombie Arena game or the Thomas Was Late game
// We can use getPosition()
Listener::setPosition(m_Thomas.getPosition().x,
    m_Thomas.getPosition().y, 0.0f);
```

The preceding code will make all the sounds play relative to that location. This is just what we need for the distant roar of a fire tile or incoming zombie, but for regular sound effects like jumping, this is a problem. We could start handling an emitter for the location of the player, but SFML makes things simple for us. Whenever we want to play a "normal" sound, we simply call `setRelativeToListener`, as shown in the following code, and then play the sound in the exact same way as we have done so far.

Here is how we might play a "normal" unspatialized jump sound effect:

```
jumpSound.setRelativeToListener(true);
jumpSound.play();
```

All we need to do is call `Listener::setPosition` again before we play any spatialized sounds and this will set the "ears" for the current sound.

We now have a wide repertoire of SFML sound functions, and we are ready to make some spatialized noise for real.

Upgrading the SoundEngine class

Let's add some new functionality to the `SoundEngine` class and start adding the spatialization features for real.

The first addition to the `SoundEngine` class is some new member variables. In `SoundEngine.h` add these two members to the `private` section:

```
    static SoundBuffer mFireballLaunchBuffer;
     static Sound mFireballLaunchSound;
```

We now have a new `SoundBuffer` to load a sound into and a new `Sound` instance to associate with the `SoundBuffer` instance and play the sound. There is nothing new here except to remember that the sound we load into `mFireballLaunchBuffer` must be mono for the spatialization to work.

Next, add the following function declaration to the `public` section of `SoundEngine.h`:

```
    static void playFireballLaunch(
    Vector2f playerPosition,
    Vector2f soundLocation);
```

The preceding code `playFireballLaunch` function declaration takes a `Vector2f` for the player's location and a `Vector2f` for the location where we want to simulate the sound coming from.

In `SoundEngine.cpp` add the following highlighted declarations before the `SoundEngine` constructor as shown and highlighted below:

```
    SoundBuffer SoundEngine::m_ClickBuffer;
    Sound SoundEngine::m_ClickSound;
    SoundBuffer SoundEngine::m_JumpBuffer;
    Sound SoundEngine::m_JumpSound;
    SoundBuffer SoundEngine::mFireballLaunchBuffer;
    Sound SoundEngine::mFireballLaunchSound;
```

The preceding code makes the static variables from `SoundEngine.h` available in `SoundEngine.cpp`. Static variables are class-owned variables and are not unique to each instance. This is just what we want because we don't want different parts of the rest of our code using different `Sound` or `Music` instances.

Now add the following initializations before the closing curly brace of the constructor in the `SoundEngine.cpp` file:

```
Listener::setDirection(1.f, 0.f, 0.f);
Listener::setUpVector(1.f, 1.f, 0.f);
Listener::setGlobalVolume(100.f);
mFireballLaunchBuffer.loadFromFile(
"sound/fireballLaunch.wav");
mFireballLaunchSound.setBuffer(
mFireballLaunchBuffer);
```

In the preceding code, we set up the `Listener` instance values for direction, set up the vector vector, and the global volume. These values are global and affect all sounds.

Add the `playFireballLaunch` function, as shown next, to the `SoundEngine.cpp` file:

```
void SoundEngine::playFireballLaunch(
  Vector2f playerPosition,
  Vector2f soundLocation)
{
  mFireballLaunchSound.setRelativeToListener(true);

  if (playerPosition.x > soundLocation.x)
  // coming from the left
  {
    Listener::setPosition(0, 0, 0.f);
    mFireballLaunchSound.setPosition(-100, 0, 0.f);
    mFireballLaunchSound.setMinDistance(100);
```

```
      mFireballLaunchSound.setAttenuation(0);
    }
    else// coming from the right
    {
      Listener::setPosition(0, 0, 0.f);
      mFireballLaunchSound.setPosition(100, 0, 0.f);
      mFireballLaunchSound.setMinDistance(100);
      mFireballLaunchSound.setAttenuation(0);
    }

    mFireballLaunchSound.play();
  }
```

In the preceding code, we call `setRelativeToListner` and pass in `true`. This is required for the spatialized sound to work. Next, we have an `if-else` structure. The condition of the `if` block determines whether or not the sound should come from the left by seeing if the player's horizontal coordinate is greater than the horizontal coordinate of the fireball that has called the function.

In both the `if` and `else` blocks, the player's horizontal position is set to `0`, the minimum distance is set to `100`, and the attenuation to `0`. What is different between the blocks is that the `setPosition` function is called with a horizontal value of `100` if the sound must come from the left, and `-100` if the sound must come from the right.

After the `if-else` structure, we finally call the `play` function on `mFireballLaunchSound`. We will call this `playFireballLaunch` function soon.

Now we have added a spatialized sound, we can code a couple of classes to represent the fireballs in our game that will use the new sound.

Fireballs

Now let's use our new function by coding the fireballs. To get started with creating our fireballs, we need some new classes. Create two new classes, `FireballUpdate` and `FireballGraphics`, and derive them from `Update` and `Graphics` respectively.

Coding the FireballUpdate class

To begin the process of coding the `FireballUpdate` class, in `FireballUpdate.h` add the following code:

```
#pragma once
#include "Update.h"
#include <SFML/Graphics.hpp>
using namespace sf;
class FireballUpdate :
    public Update
{
private:
    FloatRect m_Position;
    FloatRect* m_PlayerPosition;
    bool* m_GameIsPaused = nullptr;
    float m_Speed = 250;
    float m_Range = 900;
    int m_MaxSpawnDistanceFromPlayer = 250;
    bool m_MovementPaused = true;
    Clock m_PauseClock;
    float m_PauseDurationTarget = 0;
    float m_MaxPause = 6;
    float m_MinPause = 1;
    //float mTimePaused = 0;
bool m_LeftToRight = true;
public:
    FireballUpdate(bool* pausedPointer);
    bool* getFacingRightPointer();
    FloatRect* getPositionPointer();
    int getRandomNumber(int minHeight,
 int maxHeight);
    // From Update : Component
```

```
  void update(float fps) override;
      void assemble(shared_ptr<LevelUpdate> levelUpdate,
          shared_ptr<PlayerUpdate> playerUpdate)
   override;
  };
```

In the `private` section of the `FireBallUpdate` header file, we declare `m_Position`, `m_PlayerPosition`, and `m_GameIsPaused`, which are for the position of the fireball, a pointer to the player position, and a pointer to the `LevelUpdate` Boolean, which controls if the game is paused.

The floating-point `m_Speed` and `m_Range` variables will be initialized with random values to determine how fast the fireball will travel and how far away the fireball will begin its journey.

The `m_MaxSpawnDistanceFromPlayer` integer is set to `250` as the upper limit for how far away the fireball can spawn from the player.

The Boolean `m_MovementPaused` will work in conjunction with `m_GameIsPaused` to stop and restart the fireball in sync with the game pausing, resuming, starting, and ending.

The `Clock` instance, `m_PauseClock`, counts the time before a fireball is launched based on the random value assigned to the `float m_PauseDurationTarget`. This adds extra randomness and variation between all the fireball instances.

The `float m_MaxPause` and the `float m_MinPause` variables are fixed values in between which the random pause time will be generated, in seconds.

The Boolean `m_LeftToRight` will be flipped between `true` and false and will determine if the fireball comes from the left of the player or the right.

In the `public` section, we have the following variables and functions:

- The constructor receives a Boolean pointer that allows the fireball to track when the game is paused.
- The `getFacingRightPointer` function returns a pointer to track which way the fireball is heading. This will be shared with the `FireballGraphics` class so that it can draw the flames with the correct heading.
- The `getPositionPointer` function returns a pointer to the position of the fireball. This will be shared with the `FireballGraphics` class so it can draw the flames in the correct position.
- The `getRandomNumber` function takes two integer values and returns a random number somewhere in between.
- And, as usual, we have our two overridden functions from the `Update` class, which are `update` and `assemble`.

We will break `FireballUpdate.cpp` it into a few parts because it is quite long. For the first part add the following to `FireballUpdate.cpp`:

```
#include "FireballUpdate.h"
#include <random>
#include "SoundEngine.h"
#include "FireballUpdate.h"
#include "PlayerUpdate.h"
FireballUpdate::FireballUpdate(bool* pausedPointer)
{
  m_GameIsPaused = pausedPointer;
  m_PauseDurationTarget = getRandomNumber(m_MinPause,
m_MaxPause);
}
bool* FireballUpdate::getFacingRightPointer()
{
  return &m_LeftToRight;
}
FloatRect* FireballUpdate::getPositionPointer()
{
  return &m_Position;
```

```
}
void FireballUpdate::assemble(
  shared_ptr<LevelUpdate> levelUpdate,
  shared_ptr<PlayerUpdate> playerUpdate)
{

  m_PlayerPosition =
    playerUpdate->getPositionPointer();
  m_Position.top = getRandomNumber(
    m_PlayerPosition->top - m_MaxSpawnDistanceFromPlayer,
    m_PlayerPosition->top + m_MaxSpawnDistanceFromPlayer);
  m_Position.left =
    m_PlayerPosition->left - getRandomNumber(200, 400);
  m_Position.width = 10;
  m_Position.height = 10;
}
```

In the preceding code for the `FireballUpdate` constructor, `m_GameIsPaused` is synchronized to the variable that determines whether the game is paused within the `LevelUpdate` class and the `m_PauseDurationTarget` variable is randomly initialized. As each instance is initialized randomly, the fireballs will be at different times and won't fire at the player in one deadly wall of fireballs.

In the `getFacingRightPointer` function, the address of the `m_LeftToRight` variable is returned. The `FireBallGraphics` class will use this function to keep track of which way around they should draw the fireball texture.

In the `getPositionPointer` function, the address of `m_Position FloatRect` is returned. The `FireballGraphics` class will use this function to keep track of where the fireball is and make sure to initialize the vertices of the `VertexArray` in the correct place in the world.

In the `assemble` function, the player position address is initialized because the fireball will detect when it has hit a player and move the player accordingly. The top and left positions of the fireball are initialized using the

current position of the player (for fairness) and the `getRandomNumber` function (for variety within a range).

The width and height of a fireball (10 by 10 world units) are initialized in the final two lines of code in the `assemble` function.

For the second part of the `FireBallUpdate` class' implementation, add the following code to `FireballUpdate.cpp`:

```
int FireballUpdate::getRandomNumber(int minHeight, int maxHeight
{
  // Seed the random number generator with current time
  std::random_device rd;
  std::mt19937 gen(rd());
  // Define a uniform distribution for the desired range
std::uniform_int_distribution<int>
 distribution(minHeight, maxHeight);
  // Generate a random height within the specified range
int randomHeight = distribution(gen);
  return randomHeight;
}
```

In the preceding `getRandomNumber` function code, we have the same code that we implemented for our `LevelUpdate` class's `random` function. It returns a random number between the two values passed in.

Finally, for the `FireballUpdate` class, we will code the update function. Add the following code to `FireballUpdate.cpp`:

```
void FireballUpdate::update(float fps)
{
  if (!*m_GameIsPaused)
  {
    if (!m_MovementPaused)
    {
      if (m_LeftToRight)
```

```
        {
          m_Position.left += m_Speed * fps;
          if (m_Position.left –
            m_PlayerPosition->left > m_Range)
          {
            m_MovementPaused = true;
            m_PauseClock.restart();
            m_LeftToRight = !m_LeftToRight;
            m_Position.top = getRandomNumber(
              m_PlayerPosition->top –
              m_MaxSpawnDistanceFromPlayer,
              m_PlayerPosition->top +
              m_MaxSpawnDistanceFromPlayer);
            m_PauseDurationTarget =
              getRandomNumber(m_MinPause, m_MaxPause);
          }
        }
        else
        {
          m_Position.left -= m_Speed * fps;
          if (m_PlayerPosition->left –
            m_Position.left > m_Range)
          {
            m_MovementPaused = true;
            m_PauseClock.restart();
            m_LeftToRight = !m_LeftToRight;
            m_Position.top = getRandomNumber(
              m_PlayerPosition->top –
              m_MaxSpawnDistanceFromPlayer,
              m_PlayerPosition->top +
              m_MaxSpawnDistanceFromPlayer);
            m_PauseDurationTarget =
              getRandomNumber(m_MinPause, m_MaxPause);
          }
        }
        // Has it hit the player
if (m_PlayerPosition->intersects(m_Position))
        {
          // Knock the player down
          m_PlayerPosition->top =
            m_PlayerPosition->top +
            m_PlayerPosition->height * 2;
        }
```

```
      }
      else
      {
        if (m_PauseClock.getElapsedTime().asSeconds() >
          m_PauseDurationTarget)
        {
          m_MovementPaused = false;
          SoundEngine::playFireballLaunch(
            m_PlayerPosition->getPosition(),
            m_Position.getPosition());
        }
      }
    }
  }
```

The preceding code is long, so let's split it up into five sections and go through it a bit at a time.

In the first section, we see the following code:

```
if (!*m_GameIsPaused)
  {
    if (!m_MovementPaused)
    {
```

In the code immediately above, we check that the game is not paused and that the fireball is not pausing before it sets off on a new run.

In the second section, we see the following code:

```
if (m_LeftToRight)
{
  m_Position.left += m_Speed * fps;
  if (m_Position.left -
    m_PlayerPosition->left > m_Range)
  {
    m_MovementPaused = true;
    m_PauseClock.restart();
```

```
      m_LeftToRight = !m_LeftToRight;
      m_Position.top = getRandomNumber(
        m_PlayerPosition->top -
        m_MaxSpawnDistanceFromPlayer,
        m_PlayerPosition->top +
        m_MaxSpawnDistanceFromPlayer);
        m_PauseDurationTarget =
          getRandomNumber(m_MinPause, m_MaxPause);
    }
  }
```

In the second section above, all the code is wrapped in an `if` statement that checks that the fireball is traveling from left to right. If the fireball is traveling to the right, the fireball's position is updated according to the speed and the time elapsed since the previous update. The next `if` statement (inside the one that checked the direction of travel) checks if the fireball is in excess of `m_Range` away from the player. If it is, then it is time to pause the fireball, restart the clock, switch the direction of travel, and choose a new random height and a new random pause duration. Now the fireball is primed to fly back in the opposite direction after the `m_PauseDuration` has elapsed.

In the third section, we see the following code:

```
else
{
m_Position.left -= m_Speed * fps;
  if (m_PlayerPosition->left -
    m_Position.left > m_Range)
  {
    m_MovementPaused = true;
    m_PauseClock.restart();
    m_LeftToRight = !m_LeftToRight;
    m_Position.top = getRandomNumber(
      m_PlayerPosition->top -
      m_MaxSpawnDistanceFromPlayer,
      m_PlayerPosition->top +
      m_MaxSpawnDistanceFromPlayer);
```

```
      m_PauseDurationTarget =
        getRandomNumber(m_MinPause, m_MaxPause);
    }
  }
```

In the third section above, almost the exact same thing happens as in the preceding `if` statement. The only difference is that the fireball is moved from left to right, and when it is far enough away from the player, it is prepared to once again fly from right to left.

In the fourth section, we see the following code:

```
// Has it hit the player
if (m_PlayerPosition->intersects(m_Position))
{
// Knock the player down
  m_PlayerPosition->top =
    m_PlayerPosition->top +
    m_PlayerPosition->height * 2;
}
```

In the fourth section above, the code tests whether the fireball has hit the player. If it has hit the player, the player is knocked downwards by twice the height of the player. This is likely to cause the player to have to boost to get back on the platforms and thus let the disappearing platforms catch up a bit.

In the fifth section, we see the following code:

```
else
{
if (m_PauseClock.getElapsedTime().asSeconds() >
    m_PauseDurationTarget)
  {
    m_MovementPaused = false;
    SoundEngine::playFireballLaunch(
      m_PlayerPosition->getPosition(),
```

```
            m_Position.getPosition());
        }
    }
```

In the fifth section above, the `else` block only executes when the preceding `if` section does not. Within the `else` section, another `if` statement checks whether the elapsed time of `m_PausedClock` is greater than the randomly generated `m_PauseDuration`. If it is, then `m_MovementPaused` is set to `false`, and the fireball will be ready for a new run at the player. As a warning to the player, the `playFireBallLaunch` function is called, passing in the necessary parameters for the `SoundEngine` class to play a sound coming from the appropriate direction.

Coding the FireballGraphics class

In this section, we will code the `FireballGraphics` class. To understand the code that follows it, will help to see the graphics from the texture atlas again:

Figure 20.1: Three fireball frames

We can see that there are three frames of animation from left to right. This is perfect for use with our already coded `Animator` class. Also, as with the `PlayerGraphics` class, we will need to flip the pixels in the textures so that they face the other way when the fireball is heading from right to left. Technically speaking, we should also reverse the animations as well, but this

doesn't make much difference with fire animations, although it did make a difference with the character animations and prevented the Michael Jackson effect.

Coding FireballGraphics.h

In `FireballGraphics.h`, add the following code:

```
#pragma once
#include "Graphics.h"
class Animator;
class PlayerUpdate;
class FireballGraphics :
    public Graphics
{
private:
    FloatRect* m_Position;
    int m_VertexStartIndex;
    bool* m_FacingRight = nullptr;
    Animator* m_Animator;
    IntRect* m_SectionToDraw;
    std::shared_ptr<PlayerUpdate> m_PlayerUpdate;
public:
    // From Graphics : Component
void draw(VertexArray& canvas) override;
    void assemble(VertexArray& canvas,
        shared_ptr<Update> genericUpdate,
        IntRect texCoords) override;
};
```

To begin, we added the necessary `include` directives and forward declarations for the `Animator` and `PlayerUpdate` classes so that we can refer to them in this code file. Next, we have all the private declarations.

The `FloatRect` pointer called `m_Position` holds the position. The integer `m_VertexStartIndex`, as with all our `Graphics` derived classes, will hold the position of the first vertex in the `VertexArray`.

The Boolean pointer `m_FacingRight` will hold the address of the Boolean from the `FireballUpdate` class that determines which direction the fireball is heading.

The `Animator` instance will handle looping through the three frames of animation associated with the fireball, and `IntRect. m_SectionToDraw` will hold the texture coordinates of the current frame of animation.

The `shared_ptr<PlayerUpdate>` called `m_PlayerUpdate` will allow the `FireballGraphics` class to call all the public functions of the `FireballUpdate` class.

Next, we have all the public declarations, but we only need the two overridden functions of `assemble` and `draw`. I won't waste time going through the parameters again; it is much more interesting to see what happens inside these functions.

Coding FireballGraphics.cpp

We will code `FireballGraphics.cpp` it in a couple of stages. First, add the code for the `include` directives and the `assemble` function:

```
#include "FireballGraphics.h"
#include "Animator.h"
#include "FireballUpdate.h"
void FireballGraphics::assemble(
    VertexArray& canvas,
    shared_ptr<Update> genericUpdate,
    IntRect texCoords)
{
    shared_ptr<FireballUpdate> fu =
        static_pointer_cast
        <FireballUpdate>(genericUpdate);
    m_Position = fu->getPositionPointer();
    m_FacingRight = fu->getFacingRightPointer();
   m_Animator = new Animator(
```

```
            texCoords.left,
            texCoords.top,
            3,// 6 frames
            texCoords.width * 3,
            texCoords.height,
            6);// FPS
    // Get the first frame of animation
        m_SectionToDraw =
            m_Animator->getCurrentFrame(false);
        m_VertexStartIndex =
            canvas.getVertexCount();
        canvas.resize(canvas.getVertexCount() + 4);
        const int uPos = texCoords.left;
        const int vPos = texCoords.top;
        const int texWidth = texCoords.width;
        const int texHeight = texCoords.height;
        canvas[m_VertexStartIndex].texCoords.x =
            uPos;
        canvas[m_VertexStartIndex].texCoords.y =
            vPos;
        canvas[m_VertexStartIndex + 1].texCoords.x =
            uPos + texWidth;
        canvas[m_VertexStartIndex + 1].texCoords.y =
            vPos;
        canvas[m_VertexStartIndex + 2].texCoords.x =
            uPos + texWidth;
        canvas[m_VertexStartIndex + 2].texCoords.y =
            vPos + texHeight;
        canvas[m_VertexStartIndex + 3].texCoords.x =
            uPos;
        canvas[m_VertexStartIndex + 3].texCoords.y =
            vPos + texHeight;
    }
```

In the preceding code, we start by casting an `Update` instance into a `FireballUpdate` instance and calling the `getPositionPointer` and `getFacingRightPointer` functions so that we can track where the fireball is in the world and which direction it is facing.

Next, we initialize the `Animator` instance and initialize the starting frame's texture coordinates by calling `getCurrentFrame`. The rest of the code is as we have seen in all the other `Graphics` derived classes. We save the starting index of the quad, expand the `VertexArray` to hold the quad, and initialize the starting texture coordinates of all the vertices in the `VertexArray`.

Finally, add the `draw` function:

```
void FireballGraphics::draw(VertexArray& canvas)
{
    const Vector2f& position =
        m_Position->getPosition();
    const Vector2f& scale =
        m_Position->getSize();
    canvas[m_VertexStartIndex].position =
        position;
    canvas[m_VertexStartIndex + 1].position =
        position + Vector2f(scale.x, 0);
    canvas[m_VertexStartIndex + 2].position =
        position + scale;
    canvas[m_VertexStartIndex + 3].position =
        position + Vector2f(0, scale.y);
    if (*m_FacingRight)
    {
        m_SectionToDraw =
            m_Animator->getCurrentFrame(false);
        const int uPos = m_SectionToDraw->left;
        const int vPos = m_SectionToDraw->top;
        const int texWidth = m_SectionToDraw->width;
        const int texHeight = m_SectionToDraw->height;
        canvas[m_VertexStartIndex].texCoords.x =
            uPos;
        canvas[m_VertexStartIndex].texCoords.y =
            vPos;
        canvas[m_VertexStartIndex + 1].texCoords.x =
            uPos + texWidth;
        canvas[m_VertexStartIndex + 1].texCoords.y =
            vPos;
        canvas[m_VertexStartIndex + 2].texCoords.x =
```

```
            uPos + texWidth;
        canvas[m_VertexStartIndex + 2].texCoords.y =
            vPos + texHeight;
        canvas[m_VertexStartIndex + 3].texCoords.x =
            uPos;
        canvas[m_VertexStartIndex + 3].texCoords.y =
            vPos + texHeight;
    }
    else
    {
        // Doesn't make much difference to
// fire which order the frames are drawn
// But front must be at front duh!!!!
        m_SectionToDraw = m_Animator->getCurrentFrame(true);
  // reversed
const int uPos = m_SectionToDraw->left;
        const int vPos = m_SectionToDraw->top;
        const int texWidth = m_SectionToDraw->width;
        const int texHeight = m_SectionToDraw->height;
        canvas[m_VertexStartIndex].texCoords.x =
            uPos;
        canvas[m_VertexStartIndex].texCoords.y =
            vPos;
        canvas[m_VertexStartIndex + 1].texCoords.x =
            uPos - texWidth;
        canvas[m_VertexStartIndex + 1].texCoords.y =
            vPos;
        canvas[m_VertexStartIndex + 2].texCoords.x =
            uPos - texWidth;
        canvas[m_VertexStartIndex + 2].texCoords.y =
            vPos + texHeight;
        canvas[m_VertexStartIndex + 3].texCoords.x =
            uPos;
        canvas[m_VertexStartIndex + 3].texCoords.y =
            vPos + texHeight;
    }
}
```

We can break the preceding code into three simple sections: the starting section, the `if` block section, and the `else` block section.

In the starting section, the vertex positions are updated. They will be moving in most frames of the game except when the game is paused, or when the fireball is waiting for the random interval to end before zooming off again.

The `if` section tests if the fireball is facing right. If it is, the texture coordinates are assigned to the appropriate vertices. If the `else` section executes the texture coordinates are assigned to the vertices and horizontally flipped so the fireball is facing to the left.

Next, we will use our two new classes.

Building some fireballs in the factory

In this section, we will add code to the `Factory` class to instantiate some fireballs in the game. Add two new `include` directives to `Factory.cpp`:

```
#include "FireballGraphics.h"
#include "FireballUpdate.h"
```

Add more code to the factory after the platforms but before the rain-related code as shown next:

```
// Fireballs
for (int i = 0; i < 12; i++)
{
  GameObject fireball;
  shared_ptr<FireballUpdate> fireballUpdate =
    make_shared<FireballUpdate>(
      levelUpdate->getIsPausedPointer());
  fireballUpdate->assemble(levelUpdate, playerUpdate);
  fireball.addComponent(fireballUpdate);
  shared_ptr<FireballGraphics> fireballGraphics =
    make_shared<FireballGraphics>();
  fireballGraphics->assemble(canvas,
    fireballUpdate,
```

```
      IntRect(870, 0, 32, 32));
    fireball.addComponent(fireballGraphics);
    gameObjects.push_back(fireball);
  }
  //end fireballs
```

In the preceding code, a `for` loop iterates 12 times. Each time through the loop a fireball is created and a `GameObject` is added to the `gameObjects` vector.

The usual procedure for creating each instance is followed:

1. Create a `GameObject` instance.
2. Create an `Update` derived shared pointer adding any required constructor parameters to the call to `new`.
3. Call the `assemble` function.
4. Add the `Update` derived instance to the `GameObject` with the `addComponent` function.
5. Repeat for the `Graphics` derived shared pointer.

We are ready to see our fireballs in action!

Running the code

Now our fireballs are ready! Run the game and gaze in awe (but not forgetting to use the radar and the spatialized sound to keep out the way) at the fireballs we have coded.

Figure 20.2: Fireballs

We can also hear the directional sound. The sound tells you where a fireball is coming from, and the minimap warns you in advance if you need to get out of the way.

Summary

In this chapter, we have learned what spatialization is and that it enables us to add direction to sound in our games. We went on to learn the theory of how SFML handles spatialization. We then upgraded the `SoundEngine` class to make spatialized noises and finally coded the fireball-related classes (`Graphics` and `Update` derived) that make spatialized sounds and launch

some fireballs in the game. In the next chapter, we will add a cool parallax background and a somewhat stunning shader effect.

21

Parallax Backgrounds and Shaders

This is the last chapter where we will be working on our game. It will be fully playable with all the features by the end. Here is what we will do to wrap up the Run game:

- Learn about **OpenGL**, shaders, and the **Graphics Library Shading Language** (**GLSL**)
- Finish the `CameraGraphics` class by implementing a scrolling background and a shader
- Code a shader for the game by using someone else's code
- Run the completed game

The code for the completion of this chapter can be found in the `Run7` folder. Let's get started by learning about the OpenGL, shaders and GLSL.

Learning about OpenGL, shaders, and GLSL

The **Open Graphics Library** (**OpenGL**) is a programming library that handles 2D as well as 3D graphics. OpenGL works on all major desktop

operating systems, and there is also a version that works on mobile devices, known as **OpenGL ES**.

OpenGL was originally released in 1992. It has been refined and improved over more than twenty years. Furthermore, graphics card manufacturers design their hardware to make it work well with OpenGL. The point of mentioning this is not to give you a history lesson but to explain that it would be a fool's errand to try and improve upon OpenGL and use it in 2D (and 3D games) on the desktop, especially if we want our game to run on more than just Windows, which is the obvious choice. We are already using OpenGL because SFML uses OpenGL.

Shaders are programs that run on the GPU itself. We'll find out more about them in the following section.

The programmable pipeline and shaders

Through OpenGL, we have access to what is called a **programmable pipeline**. The programmable pipeline enables us to send our shader programs off to be drawn, each frame, with the `RenderWindow` instance's `draw` function. We can also write code that runs on the GPU that can manipulate every pixel independently, after the call to `draw`. This is a very powerful feature.

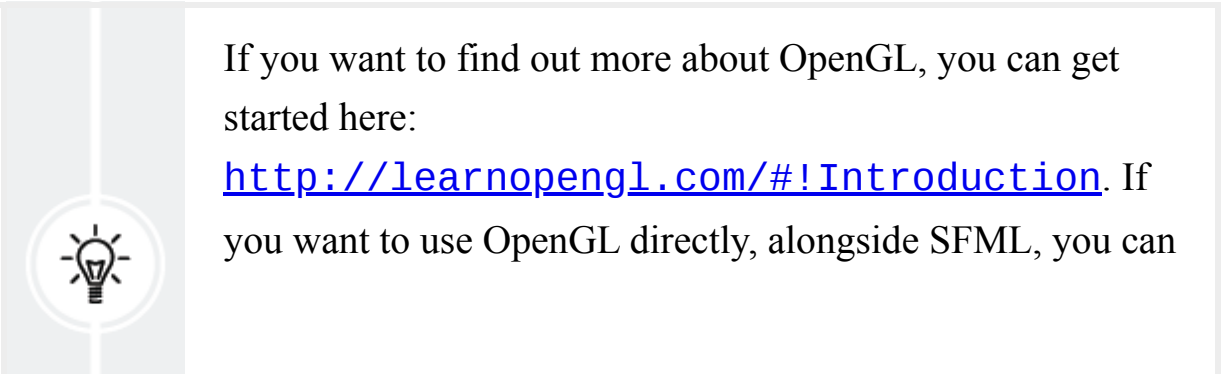
This extra code that runs on the GPU is called a **shader program**. We can write code to manipulate the geometry (position) of our graphics in a **vertex shader**. We can also write code that manipulates the appearance of every pixel individually in code. This is known as a **fragment shader**. In addition,

there are other shaders, such as compute and geometry shaders, which we will leave out for this discussion.

Although we will not be exploring shaders in great depth, we will explore some relatively simple shader code using the **GL Shader Language** (**GLSL**), which is the language you need to use in this context. In our Run project, we will utilize someone else's quite complex GLSL shader code for impressive effects.

In OpenGL, everything is a point, a line, or a triangle. In addition, we can attach colors and textures to this basic geometry and combine these elements to make the complex graphics that we see in today's modern games. These are collectively known as **primitives**. We have access to the OpenGL primitives through the SFML primitives and `VertexArray`, as well as the `Sprite` and `Shape` classes.

In addition to primitives, OpenGL uses matrices. Matrices are a method and structure for performing arithmetic. This arithmetic can range from extremely simple high-school-level calculations, such as moving (translating) a coordinate, to quite complex, such as performing more advanced mathematics, for example, to convert our game world coordinates into OpenGL screen coordinates that the GPU can use. Fortunately, it is this complexity that SFML handles for us behind the scenes. SFML also allows us to handle OpenGL directly.



If you want to find out more about OpenGL, you can get started here: `http://learnopengl.com/#!Introduction`. If you want to use OpenGL directly, alongside SFML, you can

> read this article to find out more: https://www.sfml-dev.org/tutorials/2.5/window-opengl.php.

A game can have many shaders. We can then *attach* different shaders to different game objects to create the desired effects. We will only have one vertex shader in this game, and we will apply it to the background of our game in every frame using a separate `draw` call. In SFML, you attach a shader to a `draw` call, and it affects everything in the `draw` call.

However, when you see how to attach a shader to a `draw` call, it will be plain that it is trivial to have more shaders.

We will follow these steps:

1. First, we need the code for the shader that will be executed on the GPU. We will acquire that in the *Coding a shader for the game* section.
2. Then, we need to compile that code using SFML C++ code. Visual Studio does not compile shaders for us.
3. Finally, we need to attach the shader to the appropriate `draw` function call in the `draw` function of our game.

GLSL is a language and it also has its own types, and variables of those types, which can be declared and utilized. Furthermore, we can interact with the shader program's variables from our C++ code.

As we will see, GLSL has some syntax similarities to C++.

Coding a hypothetical fragment shader

In this section, we will explore some simple, hypothetical code. We will not add this code to our Run project. Here is the code from a relatively simple shader called `fragShader.frag`:

```
// attributes from vertShader.vert
varying vec4 vColor;
varying vec2 vTexCoord;
// uniforms
uniform sampler2D uTexture;
uniform float uTime;
void main() {
    float coef = sin(gl_FragCoord.y * 0.1 + 1 * uTime);
    vTexCoord.y +=  coef * 0.03;
    gl_FragColor = vColor * texture2D(uTexture, vTexCoord);
}
```

The first four lines (excluding comments) are the variables that the fragment shader will use, but they are not ordinary variables. The first type we can see is `varying`. `varying` is for variables that are in scope between both `shaders`. Next, we have the `uniform` variables. These variables can be manipulated directly from our C++ code. We will see how we do this soon but on a more complex shader.

In addition to the `varying` and `uniform` types, each of the variables also has a more conventional type that defines the actual data, as follows:

- `vec4` is a vector with four values.
- `vec2` is a vector with two values.
- `sampler2d` will hold a texture.
- `float` is just like a `float` data type in C++.

The code inside the `main` function is executed. If we look closely at the code in `main`, we will see each of the variables in use. Exactly what this code does is beyond the scope of the book. In summary, however, the texture coordinates (`vTexCoord`) and the color of the pixels/fragments (`glFragColor`) are manipulated by several mathematical functions and operations. Remember that this executes for each pixel involved in the `draw` function,

which is called on each frame of our game. Furthermore, be aware that `uTime` is passed in as a different value for each frame. The result would be to make the graphics exhibit a rippled effect.

Coding a hypothetical vertex shader

In this section, we will see some simple hypothetical code for a vertex shader. We will not use this code in the Run project. Here is the code from the hypothetical `vertShader.vert` file. You don't need to code this:

```
//varying "out" variables to be used in the fragment shader
varying vec4 vColor;
varying vec2 vTexCoord;
void main() {
    vColor = gl_Color;
    vTexCoord = (gl_TextureMatrix[0] * gl_MultiTexCoord0).xy;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

First of all, notice the two `varying` variables, the ones that begin with v. These are the very same variables that we manipulated back in the fragment shader. In the `main` function, the code manipulates the position of each and every vertex. How the code works is beyond the scope of this book, but there is some quite in-depth mathematics going on behind the scenes. If it interests you, then exploring GLSL further will be fascinating.

In the next section, we will see how to prepare and load a real shader program as well as pass values (varyings) to each frame. We will use a shader program way in advance of the hypothetical shaders that we have just looked at.

Finishing the CameraGraphics class

In this section, we will revisit, amend, and add to our `CameraGraphics` class. First, we will add some background and shader-related variables to our `CameraGraphics.h` file. At the end of the private section of `CameraGraphics.h`, add the following variables:

```
//For the shaders and parallax background
Shader m_Shader;
bool m_ShowShader = false;
bool m_BackgrounsAreFlipped = false;
Clock m_ShaderClock;
Vector2f m_PlayersPreviousPosition;
Texture m_BackgroundTexture;
Sprite m_BackgroundSprite;
Sprite m_BackgroundSprite2;
```

In the preceding code we have an SFML `Shader` called `m_Shader` and a Boolean called `m_ShowShader`, which we can use to track when to show the shader or not. For this game, we will flip between ten seconds of showing the shader and ten seconds of showing the parallax background.

The Boolean `m_BackgroundsAreFlipped` will be used to determine whether the texture that represents the background is horizontally reversed. We do this to seamlessly join one background image to multiple instances of itself to create a smooth scrolling effect.

`Clock m_ShaderClock` is interesting as it will be the input value for one of the shader's varying values.

The `Vector2f` called `m_PlayersPreviousPosition` will allow us to know where the player was before the last update. We will see how this is useful when we add more code to the `CameraGraphics.cpp` file.

The `Texture` instance called `m_BackgroundTexture` is a separate texture for the background image. This is entirely separate from the texture atlas that holds everything else.

The `Sprite` called `m_BackgroundSprite` is for the background image. The Sprite `m_BackgroundSprite2` is to show the reversed copy of the background.

Now in the `CameraGraphics` constructor in `CameraGraphics.cpp`, just before the end of the closing curly brace, add this new code:

```
// Initialize the background sprites
m_BackgroundTexture.loadFromFile(
  "graphics/backgroundTexture.png");
m_BackgroundSprite.setTexture(m_BackgroundTexture);
m_BackgroundSprite2.setTexture(m_BackgroundTexture);
m_BackgroundSprite.setPosition(0, -200);
// Initialize the shader
m_Shader.loadFromFile(
  "shaders/glslsandbox109644", sf::Shader::Fragment);
if (!m_Shader.isAvailable())
{
  std::cout << "The shader is not available\n";
}
m_Shader.setUniform(
  "resolution", sf::Vector2f(2500, 2500));
m_ShaderClock.restart();
```

The preceding code loads the background texture and attaches it to both of our new sprites. The first of the background sprites is positioned to fill the screen behind the player.

Next, we load the shader using the `loadFromFile` function, check that the shader is available by calling the `isAvailable` function, and set the value of the uniform variable in the shader code with the `setUniform` function. The value `resolution` corresponds to a uniform variable declared within the

shader code itself; the `Vector2f` is assigned to it and used within the shader code.

Lastly, we call `restart` on `m_ShaderClock` to set it to ticking.

Finally, in the `draw` function, just after the function call shown next:

```
m_Window->setView(m_View);
```

And just before the code shown next:

```
// Draw the time UI but only in the main camera
if (!m_IsMiniMap)
{
```

Add the code that does the drawing. I have left in the above two lines and highlighted them. All the regular formatted code in between is the new code. Add it all at once to avoid mixing up the multitude of `if`, `else`, and curly brackets, and then we will break it up and go through it in smaller chunks:

```
    m_Window->setView(m_View);
/// Background stuff
    Vector2f movement;
    movement.x = m_Position->left -
        m_PlayersPreviousPosition.x;
    movement.y = m_Position->top -
        m_PlayersPreviousPosition.y;
    if (m_BackgrounsAreFlipped)
    {
        m_BackgroundSprite2.setPosition(
            m_BackgroundSprite2.getPosition().x
            + movement.x / 6,
            m_BackgroundSprite2.getPosition().y
            + movement.y / 6);
        m_BackgroundSprite.setPosition(
            m_BackgroundSprite2.getPosition().x
```

```
            + m_BackgroundSprite2.getTextureRect().getSize().x,
            m_BackgroundSprite2.getPosition().y);
        if (m_Position->left >
            m_BackgroundSprite.getPosition().x +
            (m_BackgroundSprite.getTextureRect().getSize().x / 2
        {
            m_BackgrounsAreFlipped = !m_BackgrounsAreFlipped;
            m_BackgroundSprite2.setPosition(
                m_BackgroundSprite.getPosition());
        }
    }
    else
    {
        //cout << mBackgrounsAreFlipped << endl;
        m_BackgroundSprite.setPosition(
            m_BackgroundSprite.getPosition().x - movement.x /
            6, m_BackgroundSprite.getPosition().y + movement.y /
        m_BackgroundSprite2.setPosition(
            m_BackgroundSprite.getPosition().x +
            m_BackgroundSprite.getTextureRect().getSize().x,
            m_BackgroundSprite.getPosition().y);
        if (m_Position->left >
            m_BackgroundSprite2.getPosition().x +
            (m_BackgroundSprite2.getTextureRect().getSize().x /
        {
            m_BackgrounsAreFlipped = !m_BackgrounsAreFlipped;
            m_BackgroundSprite.setPosition(
                m_BackgroundSprite2.getPosition());
        }
    }
    m_PlayersPreviousPosition.x = m_Position->left;
    m_PlayersPreviousPosition.y = m_Position->top;
    // Set the others parameters who
//need to be updated every frame
    m_Shader.setUniform("time",
        m_ShaderClock.getElapsedTime().asSeconds());
    sf::Vector2i mousePos =
        m_Window->mapCoordsToPixel(m_Position->getPosition());
    m_Shader.setUniform("mouse",
        sf::Vector2f(mousePos.x, mousePos.y + 1000));
    if (m_ShaderClock.getElapsedTime().asSeconds() > 10)
    {
        m_ShaderClock.restart();
```

```
            m_ShowShader = !m_ShowShader;
        }
        if (!m_ShowShader)
        {
            m_Window->draw(m_BackgroundSprite, &m_Shader);
            m_Window->draw(m_BackgroundSprite2, &m_Shader);
        }
        else// Show the parallax background
        {
            m_Window->draw(m_BackgroundSprite);
            m_Window->draw(m_BackgroundSprite2);
        }
     // Draw the time UI but only in the main camera
     if (!m_IsMiniMap)
```

We will now break up that large expanse of code in the upcoming section to understand it better.

Breaking up the new draw code

In breaking up the preceding code, first, we see this:

```
/// Background stuff
Vector2f movement;
movement.x = m_Position->left -
m_PlayersPreviousPosition.x;
movement.y = m_Position->top -
   m_PlayersPreviousPosition.y;
```

The preceding code declares a `Vector2f` called movement and sets `x` and `y` values using the last position of the player in the previous frame.

Next up, we have this code:

```
if (m_BackgrounsAreFlipped)
{
```

```
            m_BackgroundSprite2.setPosition(
                m_BackgroundSprite2.getPosition().x
                + movement.x / 6,
                m_BackgroundSprite2.getPosition().y
                + movement.y / 6);
            m_BackgroundSprite.setPosition(
                m_BackgroundSprite2.getPosition().x
                + m_BackgroundSprite2.getTextureRect().getSize().x,
                m_BackgroundSprite2.getPosition().y);
            if (m_Position->left >
                m_BackgroundSprite.getPosition().x +
                (m_BackgroundSprite.getTextureRect().getSize().x / 2
            {
                m_BackgrounsAreFlipped = !m_BackgrounsAreFlipped;
                m_BackgroundSprite2.setPosition(
                    m_BackgroundSprite.getPosition());
            }
    }
```

The preceding code executes when `m_BackgroundsAreFlipped` is `true`. We will see that the next block executes when `m_BackgroundsAreFlipped` is false. In the block above, `m_BackgroundSprite2` is positioned before `m_BackgroundSprite`. Here, `m_BackgroundSprite2` has its position set based on the change in the player's position from the last frame but divided by 6. The number 6 is a "magic" number that looks nice. If you increase the number 6, the background will scroll slower, and if you decrease 6, the background will scroll faster. `mBackgroundSprite` has its position set relative to the right-hand edge of `m_BackgroundSprite2`. Finally, in the preceding code, there is an `if` statement that executes when the position of the camera exceeds the left-hand edge of `m_BackgroundSprite` added to half the texture size.

This would mean that the camera is focussing on the center of `m_BackgroundSprite`, and `mBackgroundSprite2` is not "in shot" at all. This is

the perfect time to switch which background is drawn first. Then, as the camera proceeds to the right, it will appear that the city is endless. Inside the `if` statement, the `m_BackgroundsAreFlipped` Boolean is flipped, as are the positions of the backgrounds.

Next up, we have the other part of the code that we have just discussed:

```
    else
        {
            //cout << mBackgrounsAreFlipped << endl;
            m_BackgroundSprite.setPosition(
                m_BackgroundSprite.getPosition().x - movement.x /
                6, m_BackgroundSprite.getPosition().y + movement.y /
            m_BackgroundSprite2.setPosition(
                m_BackgroundSprite.getPosition().x +
                m_BackgroundSprite.getTextureRect().getSize().x,
                m_BackgroundSprite.getPosition().y);
            if (m_Position->left >
                m_BackgroundSprite2.getPosition().x +
                (m_BackgroundSprite2.getTextureRect().getSize().x /
            {
                m_BackgrounsAreFlipped = !m_BackgrounsAreFlipped;
                m_BackgroundSprite.setPosition(
                    m_BackgroundSprite2.getPosition());
            }
        }
```

The preceding code completes the logic of the flipping background by drawing the first background before the second background until the camera is focused on the second background and it is all flipped again.

Next, we have this code:

```
        m_PlayersPreviousPosition.x = m_Position->left;
        m_PlayersPreviousPosition.y = m_Position->top;
        // Set the others parameters who need
    //to be updated every frame
```

```
        m_Shader. setUniform ("time",
            m_ShaderClock.getElapsedTime().asSeconds());
        sf::Vector2i mousePos =
            m_Window->mapCoordsToPixel(m_Position->getPosition());
        m_Shader.setUniform("mouse",
            sf::Vector2f(mousePos.x, mousePos.y + 1000));
```

In the preceding code, first, we see where we save the player's position to `m_PlayersPreviousPosition`. Remember that this is how we determine the movement of the background at the start of the `draw` function. Next, we call the `setUniform` function on our `Shader` instance and pass in the name of the uniform variable to change and the current time in seconds from our `Clock` instance. Next, we get the pixel coordinates of the mouse and pass that in to set the mouse uniform in the shader.

Next up is this code:

```
    if (m_ShaderClock.getElapsedTime().asSeconds() > 10)
        {
            m_ShaderClock.restart();
            m_ShowShader = !m_ShowShader;
        }
```

In the preceding code, we test if ten seconds have elapsed since the clock was previously reset, and if so, then we reset the clock to zero and flip the value of `m_ShowShader` to alternate between showing the shader and showing the parallax background.

Finally, for what was the biggest block of code in the game:

```
    if (!m_ShowShader)
        {
            m_Window->draw(m_BackgroundSprite, &m_Shader);
            m_Window->draw(m_BackgroundSprite2, &m_Shader);
        }
```

```
        else// Show the parallax background
        {
            m_Window->draw(m_BackgroundSprite);
            m_Window->draw(m_BackgroundSprite2);
        }
```

In the preceding code, if we are not showing the shader, we will draw the background with and without the shader.

Coding a shader for the game

The only remaining task is that the file that the shader code is attempting to load is empty. The code is publicly available, but I didn't write the code and am likely not permitted to distribute it. Visit https://glslsandbox.com/e#109644.0 and click on the show code. Copy and paste all the nearly 400 lines of code into the file `shaders/glsldandbox109644`. Be sure to leave a friendly comment or thank you to the talented shader programmer who published the code. Save the file and we are ready to go. The shader code is beyond the scope of the book.

In the next section, we will see the shader in all its glory.

Running the completed game

Run the game and enjoy the new background and fire rolling countryside effect shader, which is alternated every ten seconds.

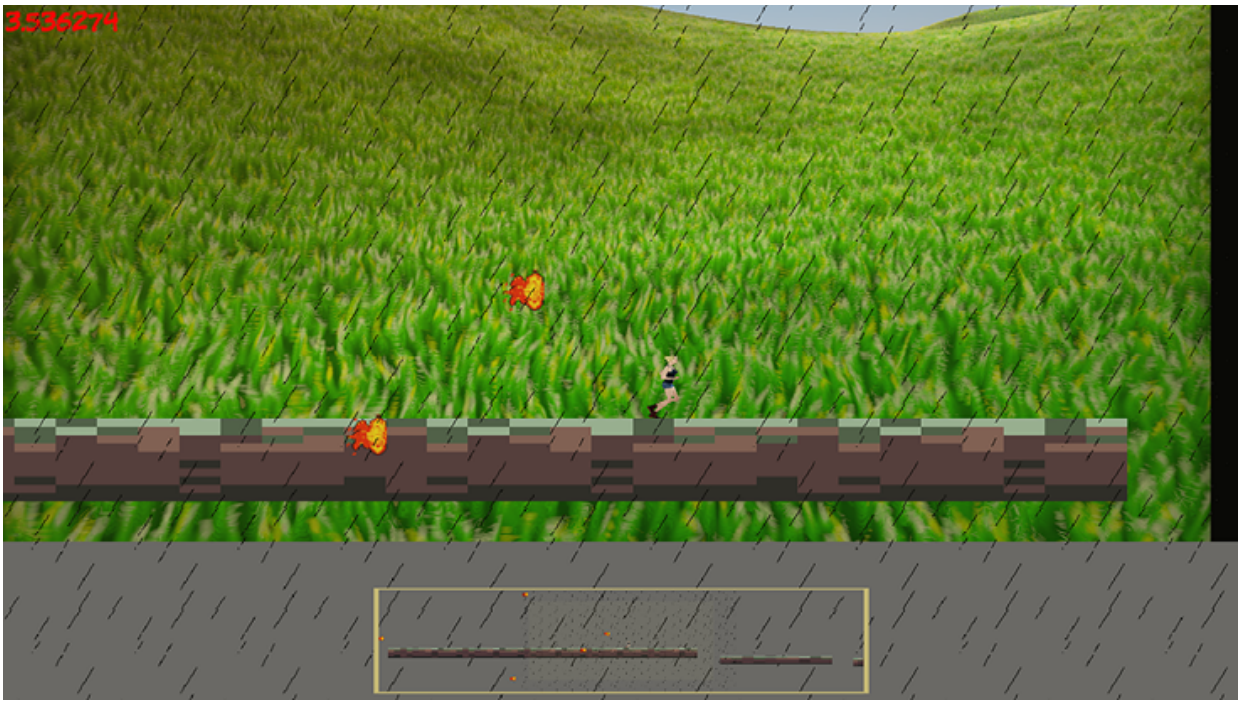


Figure 21.1: Shader

Wow! You probably agree that the capabilities of shaders and shader programmes are quite impressive.

Survive for ten seconds and the image will switch to our scrolling background, as shown in the next figure.

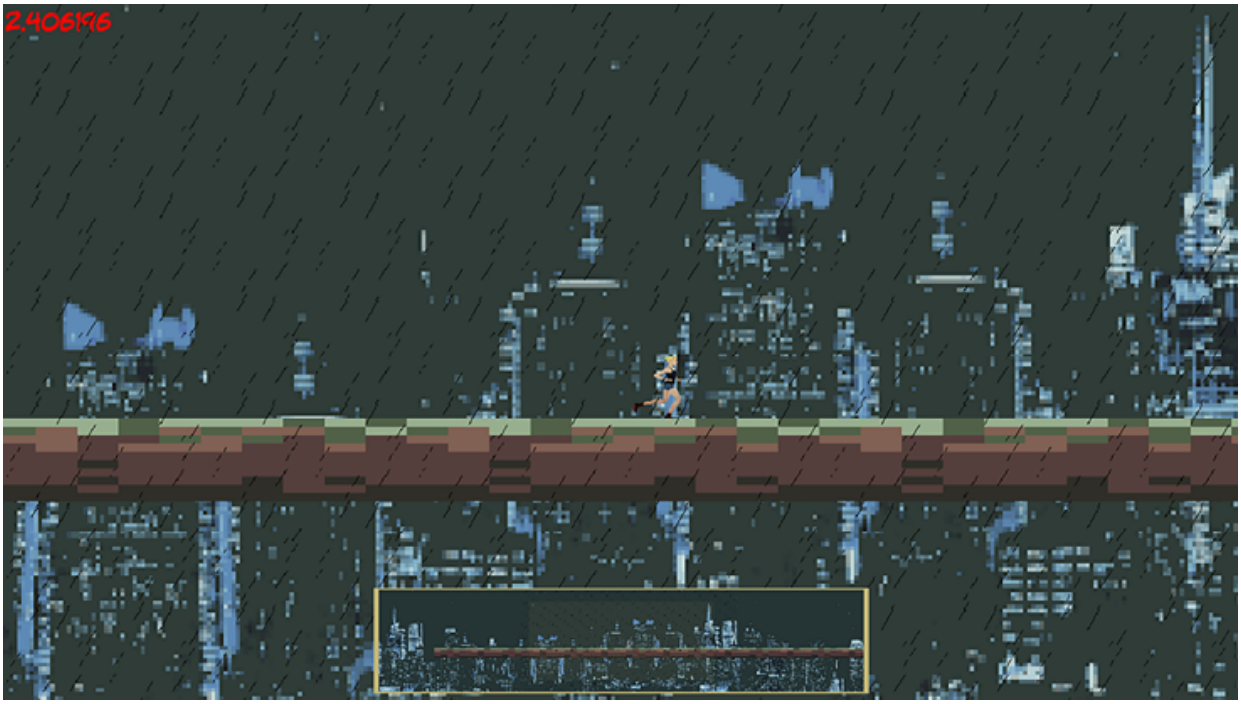


Figure 21.2: Background

That's it. Our game is completed.

Summary

When you first opened this big doorstop of a book, the back page probably seemed a long way off. But it wasn't too tough, I hope.

The point is you are here now and, hopefully, you have good insights into how to build games using C++.

The point of this section is to congratulate you on a fine achievement but also to point out that this page probably shouldn't be the end of your journey. If, like me, you get a bit of a buzz whenever you make a new game feature come to life, then you probably want to learn more.

Further reading

It might surprise you to hear that, even after all these hundreds of pages, we have only dipped our toes into C++. Even the topics we did cover could be covered in more depth, and there are numerous – some quite significant – topics that we haven't even mentioned. With this in mind, let's take a look at what might be next.

If you absolutely must have a formal qualification, then the only way to proceed is with a formal education. This, of course, is expensive and time-consuming, and I can't really help any further.

On the other hand, if you want to learn on the job, perhaps while starting work on a game you will eventually release, then what follows is a discussion of what you might like to do next.

Possibly the toughest decision we face with each project is how to structure our code. In my opinion, the absolute best source of information on how to structure your C++ game code is `http://gameprogrammingpatterns.com/`. Some of the discussion is around concepts that aren't covered in this book, but much of it will be completely accessible. If you understand classes, encapsulation, pure virtual functions, and singletons, dive into this website.

Shaders are a bigger part of game development than we have seen in this brief introduction. If you want to become an expert on shaders, I recommend reading *Anton's OpenGL 4 Tutorials* Kindle Edition available from Amazon. For some reason, the book seems suppressed slightly in the search results, and so typing the entire title into the Amazon search box might be required. It is cheaper and more comprehensive than most of the other books on the topic. It is also important to know that the way that SFML deals with shaders is different to using shaders in pure OpenGL. It would be an excellent idea to read a little bit of OpenGL (see below) or if you are sticking with SFML

(a perfectly viable strategy), read how SFML abstracts shaders here: https://www.sfml-dev.org/tutorials/2.6/graphics-shader.php.

Regarding OpenGL, there is a mountain of textbooks available. If you like videos, I can recommend *Computer Graphics with Modern OpenGL and C++* on Udemy. If you want a textbook, try *OpenGL Programming Guide* or *Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion*.

Of course, you may want to branch out and do something completely different, especially if you want to make a state-of-the-art 3D game. In this case, you should investigate something like Unreal Engine, which also uses C++; for 2D (perhaps with a bit of 3D), try the Godot engine.

I have already pointed out the SFML website throughout this book. If you haven't visited it yet, please take a look at it: http://www.sfml-dev.org/.

When you come across C++ topics you don't understand (or have never even heard of), the most concise and organized C++ tutorials can be found at http://www.cplusplus.com/doc/tutorial/. The alternative is ChatGPT. ChatGPT is great for asking things like, "Explain this code," or, "How can I make this code better/faster?"

In addition to this, there are four more SFML books you might like to look into. They are all good books but they vary greatly in who they are suitable for. Note these books are a little out of date but I think still useful. Here is a list of the books in ascending order, from most beginner-focused to most technical:

- *SFML Blueprints* by Maxime Barbier: https://www.packtpub.com/game-development/sfml-blueprints
- *SFML Game Development By Example* by Raimondas Pupius: https://www.packtpub.com/game-development/sfml-game-development-example
- *SFML Game Development* by Jan Haller, Henrik Vogelius Hansson, and Artur Moreira: https://www.packtpub.com/game-development/sfml-game-development

You also might like to consider adding life-like 2D physics to your game. SFML works perfectly with the Box2d physics engine. This URL is for the official website: http://box2d.org/. The following URL takes you to probably the best guide to using it with C++: http://www.iforce2d.net/.

If you feel like you are late to the C++ game programming party, don't worry. I thought I was late twenty-five years ago, but more C++ games are being made today than ever before. If you want to be really cutting edge, consider investigating blockchain. Blockchain enables Web3 gaming, a new type of gaming that uses blockchain technology to create a more immersive and rewarding experience for players. In Web3 games, players own their in-game assets, which can be traded or used in other games. This creates a more open and compelling gaming ecosystem. Imagine that someone plays a Pokemon game and wins a digital Pokemon card in a digital wallet app. They can then trade that card, which might be rare or even unique, with other people online or in the school playground. This is the promise of blockchain gaming. Unfortunately, most attempts so far have resulted in mediocre games and even financial scams. I bet that whoever gets this right

will create an explosion of interest in the genre. My point is that you are not late to C++ game programming. Things are only just getting started.

Most importantly, thanks very much for buying this book, and keep making games!

packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

packt
2ND EDITION
C++ Game Animation Programming
Learn modern animation techniques from theory to implementation using C++, OpenGL, and Vulkan
MICHAEL DUNSKY | GABOR SZAUER

C++ Game Animation Programming – Second Edition

Michael Dunsky

Mr. Gabor Szauer

ISBN: 978-1-80324-652-9

- Create simple OpenGL and Vulkan applications and work with shaders
- Explore the glTF file format, including its design and data structures
- Design an animation system with poses, clips, and skinned meshes
- Find out how vectors, matrices, quaternions, and splines are used in game development

- Discover and implement ways to seamlessly blend character animations
- Implement inverse kinematics for your characters using CCD and FABRIK solvers
- Understand how to render large, animated crowds efficiently
- Identify and resolve performance issues

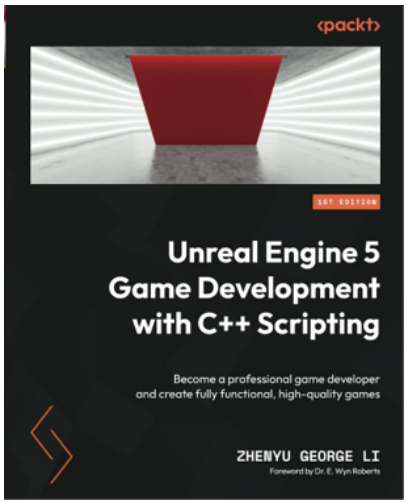
Hands-On Unity Game Development – Fourth Edition

Nicolas Alejandro Borromeo

Juan Gabriel Gomila Salas

ISBN: 978-1-83508-571-4

- Build a game that includes gameplay, player and non-player characters, assets, animations, and more
- Learn C# and Visual Scripting to customize player movements, the UI, and game physics
- Implement Game AI to build a fully functional enemy capable of detecting and attacking
- Use **Universal Render Pipeline** (**URP**) to create high-quality visuals with Unity
- Create win-lose conditions using design patterns such as Singleton and Event Listeners
- Implement realistic and dynamic physics simulations with the new Physics System

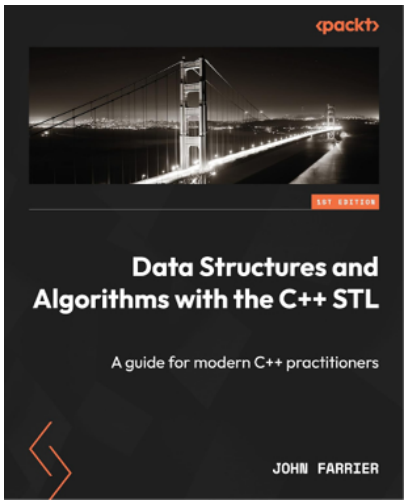


Unreal Engine 5 Game Development with C++ Scripting

Zhenyu George Li

ISBN: 978-1-80461-393-1

- Develop coding skills in Microsoft Visual Studio and the Unreal Engine editor
- Discover C++ programming for Unreal Engine C++ scripting
- Understand object-oriented programming concepts and C++-specific syntax
- Explore NPC controls, collisions, interactions, navigation, UI, and the multiplayer mechanism
- Use the predefined Unreal Engine classes and the programming mechanism
- Write code to solve practical problems and accomplish tasks
- Implement solutions and methods used in game development



Data Structures and Algorithms with the C++ STL

John Farrier

ISBN: 978-1-83546-855-5

- Streamline data handling using the std::vector
- Master advanced usage of STL iterators
- Optimize memory in STL containers
- Implement custom STL allocators
- Apply sorting and searching with STL algorithms
- Craft STL-compatible custom types
- Manage concurrency and ensure thread safety in STL
- Harness the power of parallel algorithms in STL

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Beginning C++ Game Programming, Third Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

abstract class 419, 420

address of operator 286-289

advanced OOP

inheritance 415

polymorphism 418, 419

algorithms 66

Animator class

coding 521-524

arithmetic operators 54, 55

array notation 122

arrays 121, 295

declaring 121, 122

elements, initializing 122

for games 123

arrow operator 295

assignment operators 54, 55

attenuation 560

auto keyword 303

axe

animating 165-167

axis-aligned bounding box (AABB) collision detection 212

B

background

randomly generated scrolling background, creating 273-279

using 280-282

background sprite

double buffering 42, 43

Ball class

coding 206-209

using 209, 210

Bat class

Bat.cpp, coding 195-197

Bat.h 194

Bat.h, coding 192, 193

coding 192

constructor functions 193, 194

main function, coding 198-202

using 198-202

bee

drawing 64, 65

moving 76-79

preparing 61, 62

BFXR

URL 24

Bloater 311

block 30

Boolean 333

branches 142

drawing 146

growing 142

moving 147-150

preparing 143, 144

sprites, updating in each frame 144-146

bugs 45

Bullet class

coding 332

getPosition function 340

getShape function 340

header file, coding 332-335

shoot function, coding 335-337

source file, coding 335

stop function 340

update function 340, 341

bullets

bullet array 342, 343

Bullet class, including 342

control variables 342
drawing, in each frame 347, 348
gun, reloading to shoot 343-345
making fly 341
shooting 345-347
updating, in each frame 347

C

C# 7
C++20 55, 215
C++ programming language 52
C++ game development 6
C++ references 263-267
C++ spaceship operator 215, 216
C++ standard library 383
header files 32
C++ strings 93
declaring 93
manipulating, another way with StringStream 95, 96
SFML Text and Font 96, 97
string concatenation 94
string length 94, 95
value, assigning 93
C++ variables 48

camera classes

coding 479

CameraGraphics class

draw code, breaking up 591-594

finishing 587-591

CameraGraphics class part 1

coding 484-488

CameraGraphics class part 2

coding 490-495

camera instances

adding, to game 495-498

cameras 478, 479

CameraUpdate class

coding 479-484

Chaser 311

child class 417

chopped logs

animating 165-167

class 32-34, 180-183

extending 416-418

class enumerations 126

declaring 127-129

clock.restart() function 74

clouds

adding 59
blowing 80-86
drawing 64, 65
preparing 62-64
code
creating, with comments 28
improving 175
main function 28, 29
presentation 29
syntax 29
code base 4
code files
managing 249-251
collision detection 205, 211-214, 364, 365
player and pickup collision detection 369
rectangle intersection, using 364
scoring 211-214
zombie and bullet collision detection 365-367
zombie and player collision detection 368
compile errors 44
Component polymorphic type 435
compound assignment operators 55
concrete class 419
configuration errors 44

constants 52

constructor function 192

control flow statements 129

coordinates 25

Crawler 311

crosshair

adding, for player 348-351

C-style code comments 38

C-style comment 28, 38

D

dangling pointer 294

data structures 121

death

handling 167-170

decision making

else keyword, using 69, 70

if keyword, using 69

reader challenge 71

with if and else keywords 67

decrement operator 58, 59

definition 130

delta time 74

dereference operator 286, 290

design pattern 182, 421

directive 32

double buffering 39

background sprite 42, 43

draw calls 478

E

emitter 560

encapsulation 181

entity component programming pattern 398

entity-component system pattern 422

code, running 439

Component class, coding 435

composition, preferring over inheritance 424, 425

factory pattern 426, 427

GameObject class, coding 432-434

generic GameObject, using 422, 423

Graphics class, coding 436, 437

smart pointers 428

smart pointers, casting 431

Update class, coding 438, 439

errors

bugs 45

compile errors 44

configuration errors 44
handling 44
link errors 45
expressions 55
assignment 55-57
extended Soldier class 417
extraction operator 119

F

factory class
coding, to use new classes 470
texture coordinates 470-474
factory pattern 426
FireballGraphics class
coding 574, 575
FireballGraphics.cpp
coding 578-580
FireballGraphics.h
coding 575, 576
fireballs
building, in factory 580, 581
code, running 582
coding 565
FireballUpdate class

coding 565-574
font 23
for loops 119, 120
fragment shader 584
frame rate problem 72, 73
Freesound
URL 24
function epilogue 132
function prologue 131
functions 29, 129-131, 183
function body 139
function names 137, 138
function parameters 138
function prototypes 139, 140
function return types 133-137
function scope 140
organizing 140
stack frame 131
values, returning from 30
fundamental types 50

G

game
background, drawing 40

camera instances, adding to 495-498
coding, working with 28
improving 175-177
pausing 89-92
restarting 89-92, 390
running 31, 35, 40, 214, 474, 498, 499, 512, 520, 533, 534
score and message, adding 97-103
setup handling 158, 159
time-bar, adding 104-110
game development 1, 6, 598
game logic
coding 445
game loop 36, 37
coding 251-260
input, obtaining 38
scene, drawing 38
scene, updating 38
game programming, with C++
learning, reasons 5-7
Git 9
global scope 143
GL Shader Language (GLSL) 584, 585
graphics module 193
graphics processing unit (GPU) 40

H

header file 18, 32

heads-up display (HUD) 97, 191, 219

drawing 378-381

home screen 378-381

level-up screen 378-381

objects, adding 371-375

Text objects, adding 371-375

updating 375-377

heap 292

high score

loading 383-385

saving 383-385

horde 310, 318-322

hypothetical fragment shader

coding 585, 586

hypothetical vertex shader

coding 586, 587

I

increment operator 57, 58

inheritance 181, 398, 415

initializer list 195

instance 180

integer 29

Integrated Development Environment (IDE) 8

IntelliSense 9

interactive menu

building 536

MenuGraphics class, coding 543-548

MenuUpdate class, coding 536-543

internal coordinates 25, 27

K

keyboard handling 156

key press

detecting 39

key release

detecting 163, 164

Komika Poster font 24

L

Last In, First Out (LIFO) order 132

LevelUpdate class

coding 446-459

link errors 45

Linux 10

listener 560

local coordinates 27

local variables 132, 141, 185

logical operators 67, 68

loops 114

- breaking out 117-119
- for loops 119, 120
- while loops 114-117

M

Mac 10

main function 28, 29

map 300

- data, adding 301
- data, finding 301
- data, removing 301
- declaring 300
- keys, checking 302
- key-value pairs, iterating through 302, 303
- key-value pairs, looping through 302, 303
- size, checking 301

mechanics of game 20

member access operator 295

member variables 185

memory access violation error 294

menu

building, in factory 548-550

game, running 550, 551

MenuGraphics class

coding 543-548

MenuUpdate class

coding 536-543

Mersenne Twister pseudo-random number generator 452

method 183

multi-line comments 28

N

namespace sf

using 34

non-player characters (NPCs) 33

O

object-oriented programming (OOP) 6, 32, 34, 180

class 183

need for 182

principles 180

tasks 180

object-oriented programming, principles

encapsulation 181

inheritance 181

polymorphism 181

objects 32, 34

OpenGL ES 97, 583

Open Graphics Library (OpenGL) 583

shaders 401

organizing functions 140

P

parameters 131, 185

parent class 417

Pickup class

coding 352

function definitions, coding 355-360

header file, coding 352-355

using 360-363

pixels 25

platform game 4

PlatformGraphics class

coding 508-511

platforms

building, in factory 511, 512

coding 501

PlatformUpdate class

coding 502, 504

update function, coding 504-507

player

allowing, to level up 387-390

drawing 155, 156

preparing 153-155

player animations

coding 524-533

player character

functionality, adding 513

player chopping

detecting 159-163

Player class, Zombie Arena game

building 226

function definitions, coding 233-242

header file, coding 227-232

player controls

coding 514-519

PlayerGraphics class

coding 464-469

player input

handling 156, 157

PlayerUpdate class

coding 460-464

pointers 285, 286

and arrays 295

declaring 287, 288

declaring, to object 295

dereferencing 290, 291

dynamically allocated memory 292-294

initializing 288, 289

passing, to function 294, 295

reinitializing 289, 290

syntax 286, 287

polymorphism 181, 398, 418, 419

Pong 3, 4

reference link 4

Pong bat

class, declaring 183-187

class function definitions 187, 188

functions, declaring 183-187

instance of class, using 189

reference link 184

theory 183

variables, declaring 183-187

Pong project

creating 190, 191

preprocessing 32

preprocessor 32

primitives 584

types 270

private variables 418

programmable pipeline 584

project

creating, in Visual Studio 2022 14-17

project assets 23, 223

adding 24

exploring 24, 25

sound FX, creating 24

project properties

configuring 18-20

protected specifier 418

protected variables 418

public variables 418

Python 7

R

rain

creating 551

creating, in factory 556, 557

game, running 558

RainGraphics class

coding 551-556

randomly generated scrolling background

creating 273-279

using 280-283

random number generator 66

random numbers 65

generating, in C++ 66

real game

running 43

reference variable 264

RenderWindow 34, 35

Run 4

Run project 397, 398

creating 401-403

Factory class, coding 413-415

game menu 399

game parallax 401

game rain 400

game shader 401

input handling 408-413

main function, coding 403-408

S

scope 77

scope resolution operator 188

score

adding, to game 97-103

scorekeeping 205

screen coordinates 25, 27

SFML features

including 32

SFML frame rate solution 73-76

SFML (Simple and Fast Multimedia library) 153

SFML sound

playing 171

sound code, adding 171-175

working 170

SFML vertex arrays 267

SFML VideoMode 34, 35

SFML View class 243, 478, 488, 489, 490

game camera, controlling with 243-245

reference link 381

shader 584

coding 595

shader program 584

shared smart pointers 428, 429

shoot function

coding 335-337

explanation 339
gradient, calculating 338
gradient, making positive 338
X and Y ratio, calculating 338, 339
signature 129
Simple Fast Media Library (SFML) 7
download link 12
setting up 12-14
window, opening with 31
singleton 306
smart pointers 428
casting 431
shared pointers 428, 429
unique pointers 430
Soldier class declaration 416
sound effects 23
preparing 386, 387
SoundEngine class
coding 442-445
upgrading 563-565
sounds
playing 391
shooting sound, playing 392
sound effects, adding while player reload 391

sound, playing when getting pickup 393
sound, playing when player is hit 392, 393
splat sound, playing when zombie is shot 394
spaceship operator 7, 215
spatialization 559, 560
attenuation 560
emitter 560
handling, with SFML 560-562
listener 560
sprite 27, 477
drawing 155, 156
preparing, with texture 40-42
sprite sheet 268-270
stack 132, 292
Standard Template Library (STL) 6, 285, 297, 305
auto keyword 303
list 297
map 297, 300
set 298
vector 297, 298
static class 306
static function 305
string concatenation 94
StringStream 95

strong_ordering 216

subclassed 418

superclass 417, 418

switch

decisions making with 124, 125

T

tearing 39

texture 23

used, for preparing sprite 40-42

texture coordinates 270

TextureHolder class

background texture 327, 328

function definitions, coding 307-309

header file, coding 306, 307

implementing 305, 306

Player class texture 328, 329

using 310

using, for textures 327

Texture Management 285

Timber 2

features 21

project, planning 20-23

Timberman 2

reference link 3

time-bar

adding, to game 104-110

timing 72

frame rate problem 72, 73

SFML frame rate solution 73-76

tree

drawing 64, 65

preparing 59-61

type 29, 183

U

Unified Modeling Language (UML)

URL 187

unique smart pointer 430

user-defined types 50

V

values

returning, from function 30

variables 35, 48, 183

constants 52

declaration and initialization, combining 51

declaring 50, 51

initializing 51
manipulating 54
types 48, 49
uniform initialization 52
user-defined types 50
user-defined types, declaring 53
user-defined types, initializing 53
variable scope 140
vector 298
data, accessing 299
data, adding 298
data, removing 299
declaring 298
elements, iterating through 299
elements, looping through 300
size, checking 299
version control systems (VCSs) 9
vertex 270
vertex array 270
building 271, 272
using, to draw 272, 273
vertex shader 584
View class 219
Visual Studio 8, 9

Visual Studio 2022

installing 10, 11

project, creating 14-17

W

while loops 37, 114-117

wild pointer 294

Z

Zombie Arena game 4

assets, adding to project 225

background building, from tiles 270

camera, controlling with SFML View 243, 245

engine, starting 245-249

graphical assets 224, 225

OOP skills, using 225

planning 220

Player class, building 226

project assets 223

project, creating 221-223

reference link 4

starting 220, 221

Zombie class

using, to create horde 318-322

Zombie.cpp file

coding 313-318

Zombie.h file

coding 310-312

zombies

bringing, to life 322-327

horde, building 310

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don’t worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don’t stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:

https://packt.link/free-ebook/9781835081747

2. Submit your proof of purchase.
3. That’s it! We’ll send your free PDF and other benefits to your email directly.